BEA WebLogic Enterprise

Using the idltojava Compiler

WebLogic Enterprise 5.0
Document Edition 5.0
December 1999

Using the idltojava Compiler

Document Edition	Date	Software Version
5.0	December 1999	BEA WebLogic Enterprise 5.0

Contents

About This Document

1. Overview of CORBA Java Programming

2. Using the idltojava Command

3. Java IDL Examples

4. Java IDL Programming Concepts

About This Document

This document explains what Java IDL is and describes how to use the idltojava compiler for developing Java - CORBA applications in the BEA WebLogic Enterprise (WLE) environment.

This document covers the following topics:

- Chapter 1, “Overview of CORBA Java Programming,” explains the relationship of Java IDL to CORBA, provides an overview of distributed application concepts and CORBA, and explains how you can use Java IDL to create Java applications 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. Other examples illustrate use of Persistent State and User Exceptions, Callback Objects, and Implementation Inheritance.
- Chapter 4, “Java IDL Programming Concepts,” discusses some relavant programming concepts, such as Exceptions, Initialization, and use of the Factory Finder.
- Chapter 5, “IDL to Java Mappings Used By the idltojava Compiler,” explains the CORBA IDL to Java mappings that the idltojava tool implements.
- Chapter 6, “The Java IDL API,” provides links to the Javadoc API reference pages that relate to Java IDL and the idltojava compiler.

What You Need to Know

This document is intended mainly for developers who are interested in building distributed Java applications that can act as Common Object Request Broker Architecture (CORBA) objects in a BEA WebLogic Enterprise application. It assumes a familiarity with the WebLogic Enterprise platform and Java programming.

e-docs Web Site

The BEA WebLogic Enterprise product documentation is available on the BEA 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.beasys.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 the Adobe Acrobat Reader, you can get 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 *WLE Bibliography* in the WebLogic Enterprise 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 WebLogic Enterprise documentation is important to us. Send us e-mail at **docsupport@beasys.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 5.0 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.beasys.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

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

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes

- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

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 file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\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 CORBA Java Programming

BEA WebLogic Enterprise (WLE) is an implementation of the Java 2 Enterprise Edition (J2EE) platform. As such, WLE includes CORBA (Common Object Request Broker Architecture) capability for standards-based interoperability and connectivity.

The WLE platform allows distributed Web-enabled Java applications to transparently invoke operations on remote network services using the industry standard Object Management Group (OMG) Interface Definition Language (IDL) and Internet Inter-ORBProtocol (IIOP) defined by the OMG. Runtime components include a fully compliant Java Object Request Broker (ORB) for distributed computing using IIOP communication.

To build Java applications that can access CORBA objects, you need the BEA **idltojava** compiler, a tool that converts IDL files to Java stub and skeleton files. The idltojava compiler is included with the BEA WebLogic Enterprise 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?
- About Distributed Applications
- About CORBA and Java IDL
- What's next?

Where do I get the BEA idltojava compiler?

The WebLogic Enterprise CD-ROM includes the BEA version of the idltojava compiler. Once you have installed WebLogic Enterprise, you can find the idltojava compiler in `WLEDIR/bin`.

How does the BEA idltojava compiler differ from the Sun Microsystems, Inc. version?

The BEA WLE idltojava compiler provided with WLE includes several enhancements, extensions and additions that are not included in the original compiler produced by Sun Microsystems, Inc. The WLE specific revisions are summarized here. For detailed information on using the idltojava compiler provided with WLE, see the topic “Using the idltojava Command” on page 2-1.

The WLE idltojava compiler:

- The behavior and defaults of the flags differs from that described in the Sun Microsystems, Inc. documentation. (See “idltojava Command Flags” on page 2-4.)
- Includes a new #pragma tag: `#pragma ID` *<name> <Repostitory_id>* (See “Using #pragma in IDL Files” on page 2-6.)
- Includes a new #pragma tag: `#pragma version` *<name> <m.n>* (See “Using #pragma in IDL Files” on page 2-6.)
- Extends the `#pragma prefix` to work on inner scope. A blank prefix reverts. (See “Using #pragma in IDL Files” on page 2-6.)
- 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.

What is Java IDL?

CORBA is the standard distributed object architecture developed by the OMG consortium. The OMG has specified an architecture for an ORB on which object components written by different vendors can interoperate across networks and operating systems. The OMG-specified Interface Definition Language (IDL) is used to define the interfaces to CORBA objects.

Sun Microsystems, Inc. defines “Java IDL” as:

> *The classes, libraries, and tools that make it possible to use CORBA objects from the Java programming language. The main components of Java IDL are an ORB, a naming service, and the idltojava compiler.*

Note that *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 WLE system provides its own “brand” of Java IDL. In other words, as a J2EE implementation, WLE provides all of the components you need to build Java applications capable of accessing CORBA objects. The key components in WLE are:

- WLE idltojava compiler— A tool for converting IDL interface definitions to Java stub and skeleton files
- WLE CORBA Object Request Broker (ORB)
- Java Naming and Directory Interface (JNDI)—The standard naming service available in the Java 2 Enterprise Edition (J2EE)
- Bootstrap Object and FactoryFinder—WLE objects that work in conjunction with the naming service to supply local and remote object references

About Distributed Applications

The term *distributed computing* refers to the paradigm in which data and applications in a system are spread out over multiple hosts on a network. Over time, more and more enterprises are moving applications and data to where they can operate most efficiently in the enterprise, to some mix of desktop workstations, local area network servers, regional servers, Web servers, and other servers. “Client-server” computing fits into this model in that it implies that clients will provide certain capabilities for a user and request others from server applications that provide services for the clients. The servers can be on remote machines. The Web's HTTP protocol is an example.

The e-business market that is emerging today in the context of the World Wide Web demands an object-oriented view of distributed computing as a basis for business systems.

This section discusses general concepts that relate to all distributed computing environments:

- Multi-Tiered Applications
- User Interface Tier

- Service or Business Logic Tier
- Data Store (Database) Tier

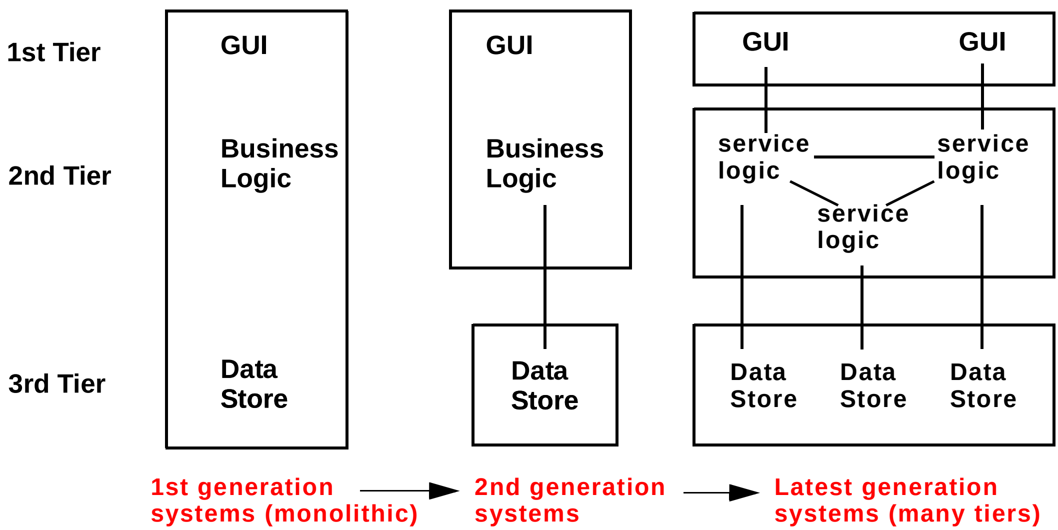
Multi-Tiered Applications

Traditional enterprise applications are, for the most part, self-contained monolithic programs with limited access to one another's procedures and data. They are usually cumbersome to build and expensive to maintain because even simple changes require the entire program to be recompiled and retested.

By contrast, applications built using distributed objects such as CORBA naturally lend themselves to a multi-tiered architecture, resulting in a useful separation of concerns.

A three-tiered application has a user interface code layer, a computation code (or business logic) layer, and a database access layer. All interaction between the layers occurs via the interfaces that all CORBA objects must publish. Figure 1-1 illustrates the transition from monolithic applications to multi-tiered, modular applications.

Figure 1-1 Transition from Monolithic to Multi-Tiered Applications



User Interface Tier

The user interface (UI) tier is the layer of user interaction. Its focus is on efficient user interface design and accessibility throughout your organization. The user interface tier can reside on the user's desktop, on your organization's intranet, or on the World Wide Web (Internet). Several user interface implementations may be deployed which access the same server. The UI tier usually invokes methods on the business logic tier and thus acts as a client of the business logic servers.

Service or Business Logic Tier

The service, or business logic layer, is server-based code with which the client code interacts. The business logic layer is made up of business objects: CORBA objects that perform logical business functions such as inventory control, budget, sales orders, and billing. These objects invoke methods on the Database tier objects.

Data Store (Database) Tier

The data store layer is made up of objects that encapsulate database routines and interact directly with the DBMS product(s). For example, a hypothetical `get_Sales_Sum method` might be implemented to obtain data from a relational database via the appropriate appropriate JDBC SQL SELECT statements.

About CORBA and Java IDL

The following sections explain more about CORBA and Java IDL:

- About CORBA
- Accessing CORBA Objects from Java Applications
- A Quick Review of CORBA Concepts

- Defining and Implementing CORBA Objects
- Client Implementation
- The FactoryFinder

About CORBA

The Common Object Request Broker Architecture (CORBA) is the standard distributed object architecture developed by the OMG Consortium. The OMG has specified an architecture for an open software bus, or Object Request Broker (ORB), on which object components written by different vendors can interoperate across networks and operating systems. This standard allows CORBA objects to invoke one another without knowing where the objects they access reside or in what language the requested objects are implemented. The OMG-specified Interface Definition Language (IDL) is used to define the interfaces to CORBA objects.

CORBA objects differ from typical programming language objects in these ways:

- CORBA objects can be located anywhere on a network.
- CORBA objects can interoperate with objects on other platforms.
- CORBA objects can be written in any programming language for which there is a mapping from OMG IDL to that language. (Mappings currently specified include Java, C++, C, Smalltalk, COBOL, and Ada.)

Accessing CORBA Objects from Java Applications

To make it possible for you to access CORBA objects from your Java applications, WLE provides the following features which are integral to the “Java IDL” programming model:

- WLE idltojava compiler—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 Chapter 2, “Using the idltojava Command.”

- WLE CORBA Object Request Broker (ORB)—The ORB together with the idltojava compiler can be used to define, implement, and access CORBA objects from Java applications. The WLE system supports both transient and persistent CORBA objects. Transient objects are those whose lifetimes are limited by their server process's lifetime. Persistent or *stateful* objects are those which can store state and reinitialize themselves from this state each time the server is restarted. (For more on using persistent objects, see the topic "Persistent State and User Exceptions IDL Example" on page 3-3 and the section on Joint Client Server Applications in Understanding Server-to-Server Communication in the WebLogic Enterprise online documentation.
- Java Naming and Directory Interface (JNDI)—The capability of looking up and locating objects in WLE is provided by the Java Naming and Directory Interface (JNDI), a standard J2EE naming service. JNDI is used along with the BEA WLE Bootstrap object and FactoryFinder to resolve object references.
- Bootstrap Object and FactoryFinder—The client application uses the Bootstrap object to obtain initial object references to key objects in a WLE domain, one of which is the FactoryFinder. The FactoryFinder, in turn, is used to locate factory objects. Factories are used to create application objects.

The WLE Java CORBA ORB supports both transient and persistent objects.

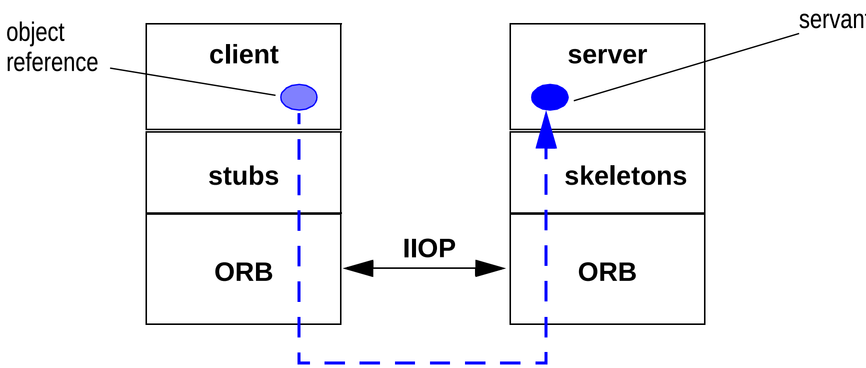
The WLE Interface Repository is not required. An interface repository is provided for dynamically determining interfaces. See the command `idl2ir` in the *Commands Reference* and *Interface Repository Interfaces* in the *CORBA Java Programming Reference* in the WebLogic Enterprise online documentation.

A Quick Review of CORBA Concepts

The concepts introduced in this section are more completely discussed in the CORBA/IIOP 2.0 Specification.

Figure 1-2 shows a method request sent from a client to a CORBA object implementation in a server. A client is any code (perhaps itself a CORBA object) that invokes a method on a CORBA object. The servant is an instance of the object implementation - the actual code and data that implements the CORBA object.

Figure 1-2 Method Request from a Client to a CORBA Object



The client of a CORBA object has an object reference for the object and the client uses this object reference to issue method requests. If the server object is remote, the object reference points to a stub function, which uses the ORB to forward invocations to the server object. The stub code uses the ORB to identify the machine that runs the server object and asks that machine's ORB for a connection to the object's server. When the stub code has the connection, it sends the object reference and parameters to the skeleton code linked to the destination object's implementation. The skeleton code transforms the call and parameters into the required implementation-specific format and calls the object. Any results or exceptions are returned along the same path.

The client has no knowledge of the CORBA object's location, implementation details, nor which ORB is used to access the object. Different ORBs communicate via the OMG-specified Internet InterORB Protocol (IIOP).

A client may only invoke methods that are specified in the CORBA object's interface, which is defined using the OMG Interface Definition Language (IDL). An interface defines an object type and specifies a set of named methods and parameters, as well as the exception types that these methods may return. An IDL compiler such as idltojava translates the CORBA object definitions into a specific programming language according to the appropriate OMG language mapping. Thus, the idltojava compiler translates IDL defintions into Java constructs according to the IDL-Java language mapping.

The stub and skeleton files are generated by the idltojava compiler for each object type. Stub files present the client with access to IDL-defined methods in the client programming language. The server skeleton files glue the object implementation to the ORB run time. The ORB uses the skeletons to dispatch methods to the object implementation instances (servants).

Defining and Implementing CORBA Objects

The goal in CORBA object development is the creation and registration of an object server, or simply server. A server is a program which contains the implementation of one or more object types and which has been registered with the ORB. For example, you might develop a desktop publishing server which implements a "Document" object type, a "Paragraph" object type, and other related object types.

CORBA Object Interfaces

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, idltojava 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 a simple IDL interface that describes the WLE sample application called `SimpApp`.

Listing 1-1 An IDL Interface for the WLE Java 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();
};
```

An operation may raise an exception when an error condition arises. The type of the exception indicates the kind of error that was encountered. Clients must be prepared to handle defined exceptions and CORBA standard exceptions for each operation in addition to normal results.

Java Language-based Implementation

After defining the IDL interfaces, the developer can build two basic types of applications with WLE:

- A remote joint client/server or client, which uses files from the idltojava command for its client stubs (and optionally also its server skeletons).

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 WLE server. Neither the client nor the server has any of the WLE administrative and infrastructure components, such as `tmadmin`, JNDI registration, and ISL/ISH (hence, none of scalability and reliability attributes of WLE)

- A server, which uses files from the m3idltojava command for its server skeletons.

The client development sequence is:

1. Define IDL interfaces for the client.
2. Run the idltojava compiler on client IDL files.
3. Implement client calls (and optionally server skeletons).
4. Compile all `.java` files into `.class` files.

5. Run the client class having a public main method which calls the WLE server and optionally also provides servants for its objects (when acting as a server).

The server development sequence is:

1. Define IDL interfaces for the server.
2. Run m3idltojava on the server IDL files.
3. Implement servant objects.
4. Compile all .java files into `.class` files.
5. Create the XML Server Descriptor File.
6. Use the `buildjavaserver` command to create a jar file.
7. Configure the JavaServer with the new jar file in a UBBCONFIG
8. Run `tmloadcf` on the `ubbconfig` file to generate a binary `tuxconfig` file
9. Run `tmboot` on the configuration file (`tuxconfig`)

An object implementation defines the behavior for all the operations and attributes of the interface it supports. There may be multiple implementations of an interface, each designed to emphasize a specific time and space trade-off, for example. The implementation defines the behavior of the interface and object creation/destruction.

Only servers can create new CORBA objects. Therefore, a factory object interface should be defined and implemented for each object type. For example, if `Document` is an object type, a `DocumentFactory` object type with a `create` method should be defined and implemented as part of the server. (Note that "create" is not reserved; any method name may be used.)

For example, here is how a WLE server registers a new object :

```
org.omg.CORBA.Object docoument_oref = TP.create_object_reference(
        DocumentHelper.id(),  // Repository ID
        docName,              // Object ID
        null                  // Routing Criteria
        );
```

The TP Framework takes cares of the actual object instantiation:

`(new DocumentServant).`

A `destroy` method may be defined and implemented on `Document` or the object may be intended to persist indefinitely. (Note that “destroy” is not reserved and any name may be used.)

The BEA Java CORBA ORB supports both transient and persistent objects. Persistent objects must be created as callback objects with the Portable Object Adapter (POA) to define a Persistent/User ID Object Policy.

Client Implementation

Client code is included on the CLASSPATH with idltojava-generated .java files and the ORB library.

Clients may only create CORBA objects via the published factory interfaces that the server provides. Likewise, a client may only delete a CORBA object if that object publishes a destruction method. A CORBA object may be shared by many clients on a network, so only the object server can know when the object has become garbage.

The client code only issues method requests on a CORBA object via the object's object reference. The *object reference* is an opaque structure which identifies a CORBA object's host machine, the port where the ISH is listening for requests, and a pointer to the specific object in the process. Because Java IDL supports only transient objects, this object reference becomes invalid if the WLE system is stopped and restarted.

Clients typically 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 refererence for `Document` to the client.

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

- 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.

The FactoryFinder

The WLE FactoryFinder interface and the NameManager give you a mechanism for registering, storing, and finding objects across multiple domains or within a single domain in WLE.

For more information on how the FactoryFinder relates to Java IDL, refer to topic “The FactoryFinder Interface” on page 4-12.

For detailed information on how to use the FactoryFinder Interface, see *FactoryFinder Interface* in the *CORBA Java Programming Reference* in the WebLogic Enterprise online documentation.

What’s next?

To get started using Java IDL to build WebLogic Enterprise Java CORBA applications, check out the following examples, concepts, and reference information:

- Using the idltojava Command
- Java IDL Examples
- Java IDL Programming Concepts
- IDL to Java Mappings Used By the idltojava Compiler
- The Java IDL API

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
- Running m3idltojava on Server Side 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 WebLogic Enterprise (WLE) idltojava compiler, see the topic "How does the BEA idltojava compiler differ from the Sun Microsystems, Inc. version?" on page 1-2

Syntax of the idltojava Command

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

`m3idltojava [`*`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 command `m3idltojava` is used to translate IDL into generic client stubs and WebLogic Enterprise server skeletons.

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, use the following command:

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

The `idltojava` command requires a C++ pre-processor, and is used to generate deprecated names. The command `idltojava` generates Java code as is appropriate for the client-side 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 WLE server. Neither the client nor the server has any of the WLE administrative and infrastructure components, such as `tmadmin`, JNDI registration, and ISL/ISH (hence, none of scalability and reliability attributes of WLE)

Running m3idltojava on Server Side IDL Files

To run `m3idltojava` on server-side IDL files, use the following command:

`m3idltojava` *<flags> <options> <idl-files>*

The server-side ORB is built to use non-deprecated names. The command `m3idltojava` generates Java code using non-deprecated names as is appropriate for the server-side ORB.

idltojava Command Options

Note: Several option descriptions have been added here that are not documented in the original Sun Microsystems Inc. idltojava compiler documentation.

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*

Option	Description
-p `package-name`	Specifies the name of an outer package to enclose all the generated Java. 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):	
-I`directory`	Specifies a directory or path to be searched for files that are *#include*d in IDL files. This option is passed to the preprocessor.
-D`symbol`	Specifies a symbol to be defined during preprocessing of the IDL files. This option is passed to the preprocessor.
-U`symbol`	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`**.

The table below includes descriptions of all flags.

Flag	Description
-f`list-flags`	Requests that the state of all the **`-f`** flags be printed. The default value of this flag is **`off`**.
-f`list -debug-flags`	Provides a list of debugger flags

`-fcaseless`	Request that case not be significant in keywords and identifiers. The default value of this flag is 'on'.
`-fclient`	Requests the generation of the client side of the IDL files supplied. The default value of this flag is `off'.
`-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 **`off`**.
`-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 `#pragma`s. 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.

Using #pragma in IDL Files

Note: The BEA WLE 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.

3 Java IDL Examples

This topic includes the following sections:

- Getting Started with a Simple Example of IDL
- Callback Objects IDL Example
- Persistent State and User Exceptions IDL Example
- Implementation Inheritance

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 WLE Java 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:

```
m3idltojava Simple.idl
```

This would generate stubs and skeletons and several other files.

For comprehensive information on how to create the Java server and client for this example, along with instructions on how to build and run it, see *The Java Simpapp Sample Application* in the *Guide to the Java Sample Applications* in the WebLogic Enterprise online documentation.

For information on the options and flags on the idltojava compiler, refer to the topic “Using the idltojava Command” on page 2-1.

Callback Objects IDL Example

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

Listing 3-2 IDL Definition for the WLE 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 *Developing Java Joint Client/Server Applications* in *CORBA Server-to-Server Communication* in the WebLogic Enterprise online documentation.

Persistent State and User Exceptions IDL Example

The WLE SimpApp example shows support of *transient* object references in WLE. If the object's server process stops and restarts, the object reference that the client is holding becomes invalid. However, Java CORBA clients can also create *persistent* object references in WLE; that is, references that remain valid even if the WLE server is stopped and restarted.

The BEA WLE system supports persistent objects by means of callbacks and the Portable Object Adapter (POA).

The POA provides transient, persistent, and other user ID policies with which to create objects in WLE. You can create a persistent object reference in WLE by creating a callback object with a Persistent/User ID Object Policy.

Implementation Inheritance

Ordinarily, servant classes must inherit from the `ImplBase` class generated by the idltojava compiler. This is inadequate for servant classes that need to inherit functionality from another Java class. The Java programming language allows a class only one superclass and the generated `ImplBase` class already occupies this position. A servant class can inherit an implementation from any Java class using Tie classes

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 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, 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 a description of minor codes thrown by the Java ORB, see “Minor Code Meanings” on page 4-4.

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. The table below lists 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.

Code	Meaning
	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 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 v1.0 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.

Code	Meaning
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 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.
	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 Java client or 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.

Creating an ORB Object

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

Applets and applications create ORB instances slightly differently, because their parameters, which must be passed in the ORB.init() call, are arranged differently.

(For more information on initializing the ORB, see the *CORBA ORB* in the *Java Programming Reference* in the WebLogic Enterprise online documentation.)

Creating an ORB for an Application

This code fragment shows how an application 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

An 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

BEA WLE uses the Sun Microsystem 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 Java 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.

The following fragment shows how a server converts a CORBA object reference to a string:

```
org.omg.CORBA.ORB orb =    // get an ORB object
org.omg.CORBA.Object obj = // create the object reference
String str = orb.object_to_string(obj);
// make the string available to the client
```

This code fragment shows how a client converts the stringified object reference back to an object:

```
org.omg.CORBA.ORB orb =    // get an ORB object
String stringifiedref =    // read string
org.omg.CORBA.Object obj = orb.string_to_object(stringifiedref);
```

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 WLE 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 four different object references (SecurityCurrent, TransactionCurrent, FactoryFinder, and InterfaceRepository). The object of concern to us here is the **FactoryFinder**.

The FactoryFinder interface provides clients with one object reference that serves as the single point of entry into the WLE 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 *Java Bootstrap Object Programming Reference* in the WebLogic Enterprise online documentation.

The FactoryFinder Interface

The FactoryFinder interface provides clients with one object reference that serves as the single point of entry into the WLE domain. The WLE NameManager provides the mapping of factory names to object references for the FactoryFinder. Multiple FactoryFinders and NameManagers together provide increased availability and reliability. Mapping across multiple domains is supported.

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

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

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

For more information about how to use the WLE FactoryFinder Interface, see the *FactoryFinder Interface* in the *CORBA Java Programming Reference* in the WebLogic Enterprise online documentation.

5 IDL to Java Mappings Used By the idltojava Compiler

The idltojava tool reads an OMG IDL interface and translates or maps it to a Java interface. idltojava 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 tool 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 attrubite
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.

6 The Java IDL API

The Java interface definition language (IDL) application programming interface (API) includes the following packages:

- com.beasys
- com.beasys.BEAWrapper
- com.beasys.Tobj
- com.beasys.TobjS
- javax.transaction
- org.omg.CosTransactions
- org.omg.Security
- org.omg.SecurityLevel1
- org.omg.SecurityLevel2

For an overview of all application programming interface (API) information related to WLE, see the *WebLogic Enterprise API Javadoc page* in the WebLogic Enterprise online documentation.

Index