BEA WebLogic Enterprise

Using RMI in a WebLogic Enterprise Environment

WebLogic Enterprise 5.0
Document Edition 5.0
December 1999

Using RMI in a WebLogic Enterprise Environment

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

Contents

3. Developing RMI Applications in WLE

4. Using RMI with Client-Side Callbacks

5. Using RMI with EJBs

6. Converting Sun JavaSoft RMI to WLE RMI Classes

7. The WebLogic Enterprise RMI API

About This Document

This document describes BEA WebLogic RMI on IIOP and explains how to develop RMI applications in a BEA WebLogic Enterprise (WLE) environment.

This document covers the following topics:

- Chapter 1, “Overview of RMI in WLE,” gives a brief introduction to remote method invocation (RMI) for distributed object systems, details the advantages of BEA WebLogic Enterprise RMI on IIOP over other flavors, discusses RMI on IIOP in terms of Enterprise JavaBeans (EJB) design considerations and the Java 2 Enterprise Edition (J2EE) platform, and gives a roadmap for working through the remainder of this guide.
- Chapter 2, “Getting Started with RMI — a Hello World Example,” takes you through a simple example of using RMI in WLE.
- Chapter 3, “Developing RMI Applications in WLE,” provides step-by-step instructions on how to develop new RMI classes for WLE, and how to build and run your new WLE application.
- Chapter 4, “Using RMI with Client-Side Callbacks,” describes how to use client-side callbacks in WLE, and in particular how this comes in handy for EJB implementations.
- Chapter 5, “Using RMI with EJBs,” explains how RMI relates to the EJB paradigm.
- Chapter 6, “Converting Sun JavaSoft RMI to WLE RMI Classes,” explains how to convert your existing Sun Microsystems JavaSoft RMI classes to function as WLE RMI objects.
- Chapter 7, “The WebLogic Enterprise RMI API,” describes the application programming interface (API) for WLE RMI.

- Appendix A, “Java Server Startup,” provides information about the use of JAR files in JavaServer startup.
- Appendix B, “Using a Startup Properties File,”explains how to use an optional startup properties file.

What You Need to Know

This document is intended mainly for developers who are interested in using BEA WebLogic Enterprise to create distributed RMI on IIOP applications that work with EJBs. 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 RMI, refer to the Sun Microsystems, Inc. Java 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`

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

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

`|`	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 RMI in WLE

This topic includes the following sections:

- What is RMI?
- What is WebLogic RMI on IIOP?
- What about RMI clients of EJBs?
- Where can I learn more about RMI?
- What software and development environment do I need for WLE RMI?
- What is next?

What is RMI?

Remote Method Invocation (RMI) is a Java based programming paradigm and application programming interface (API) for distributed object computing and Web connectivity. RMI allows an application to obtain a reference to an object that exists elsewhere on the network but then invoke methods on that object as though it existed locally on the client's Java virtual machine. So, products, services, and resources can exist anywhere on the network but appear to the programmer and the end user to be part of the local environment.

With RMI, a client object can call a remote object in a server, and that server can also be a client of other remote objects.RMI uses some form of Java serialization to marshal (encode) and unmarshal (de-code) parameters sent across a network. Serialization is a way of encoding parameters into a byte stream for delivery across a network.

What is WebLogic RMI on IIOP?

The BEA WebLogic Enterprise (WLE) development platform provides remote method invocation (RMI) as one of the standard services of a Java 2 Enterprise Edition (J2EE) implementation. For this release, WLE provides its own protocol of WebLogic RMI on the Object Management Group's industry-standard Internet Inter-Orb Protocol (IIOP). IIOP is a protocol that enables browsers and servers to exchange integers, arrays, and more complex objects, unlike HTTP which supports only transmission of text.

The primary advantage of the WLE implementation of RMI on IIOP is that it allows application developers to write remote interfaces between WLE clients and EJB servers, using a natural Java API. By making a few code enhancements, developers can also convert their legacy Java client/RMI server applications to work in WLE. WLE RMI clients can talk to EJBs and traditional RMI server objects.

WLE RMI has the following characteristics and capabilities:

- Flows over an IIOP transport—Firewalls configured to support IIOP traffic will accept WebLogic RMI on IIOP messages as standard IIOP messages.
- Uses the J2EE JNDI service for bootstrapping—In order for a client program to make a call on a remote object, the client needs to obtain a reference to the remote object. Traditional RMI uses the Java Naming Service to do this. In WLE RMI, a client gets a reference to a remote object by looking it up via the J2EE Java Naming and Directory Service (JNDI). A client can also get a reference to a remote object by receiving the reference as an argument or a return value.
- Enables clients to talk to EJBs—All clients of EJBs use RMI on IIOP
- Provides support for maintaining a transaction context between clients and remote EJBs or RMI servers—The WebLogic Enterprise system supports the Java Transaction API (JTA)

For this release, WLE IIOP does not pass Objects by Value which is needed for full CORBA interoperability support. Instead, WLE RMI on IIOP passes serialized objects as in traditional RMI. Therefore, this release of WLE does not support complete interoperability between Java clients, EJBs, and CORBA objects.

What about RMI clients of EJBs?

All clients of Enterprise Java Beans (EJB) use RMI on IIOP. Creating a client to an EJB server is essentially the same as creating an RMI client to a traditional RMI server. For more information on this, see the topic "Using RMI with EJBs" on page 5-1. For examples of clients of EJBs using RMI on IIOP to talk to EJB servers, see the *WebLogic Enterprise 5.0 EJB Sample Applications* in the WebLogic Enterprise online documentation.

Where can I learn more about RMI?

For more information about remote method invocation, refer to the The Sun Microsystems, Inc. Java site at http://java.sun.com/.

What software and development environment do I need for WLE RMI?

Before you can start developing WLE RMI applications, you need the following:

- WLE installed on your system
- Java Development Kit (JDK) 1.2 installed on your system
- Environment variables set appropriately

- CLASSPATH and PATH set to include the appropriate WLE and JDK pathnames

For information on installing WebLogic Enterprise, see the product installation guide.

For more information on setting up your development environment, see the topic “Setting Up Your WLE Development Environment” on page 3-2.

For information on the JDK, refer to the The Sun Microsystems, Inc. Java site at http://java.sun.com/.

What is next?

The following topics are covered in this guide for WebLogic Enterprise RMI on IIOP:

- Getting Started with RMI — a Hello World Example—If this is your first time using RMI, or if the WLE development environment is new to you, we suggest that you start by working through the Hello World example. This covers the basics of using remote objects in the WebLogic Enterprise environment.
- Developing RMI Applications in WLE—This section details the steps you need to follow to develop, build and run RMI applications in WLE. The Hello World example is used as a touchstone.
- Using RMI with Client-Side Callbacks
- Using RMI with EJBs
- Converting Sun JavaSoft RMI to WLE RMI Classes
- The WebLogic Enterprise RMI API

For more information about using transactions in your RMI applications, see *Using Transactions* in the WebLogic Enterprise online documentation.

2 Getting Started with RMI — a Hello World Example

This example provides a distributed version of the classic Hello World program using remote method invocation (RMI) in a WebLogic Enterprise environment.

This topic includes the following sections:

- Where can I find the RMI Hello World example?
- What is the RMI Hello World Example and what do I need to run it?
- Building and Running the Hello World Example
- Cleaning up the Directory
- Understanding the Hello World Example

Where can I find the RMI Hello World example?

In addition to the fully supported examples supplied on the CD-ROM with this release of WebLogic Enterprise (WLE), the BEA WLE team provides several *unsupported* code examples on a password protected Web site for WLE customers. You can get all

the files for the WLE RMI Hello World example from this Web site. The URL for the unsupported samples WLE Web site is specified in the product Release Notes under "About This BEA WLE Release" in the subsection "Unsupported Samples and Tools Web Page." On the samples Web page, the RMI HelloWorld example is in a directory similar to the following:

```
/unsupported/samples/rmi/helloworld
```

What is the RMI Hello World Example and what do I need to run it?

The WLE RMI Hello World example is a simple application for demonstrating remote method invocations in a distributed WebLogic Enterprise environment. The example shows a client making a remote method call to a server object running on the host. When you run the client at the command line, "Hello World!" is displayed in response.

Required Software and Environment

To run the WLE RMI Hello World example, you need WLE installed on your system and the appropriate environment variables set. The Hello World example does some automated environment setup for you, so for now the only variables you should need to check are these:

- Make sure TUXDIR is set to the full pathname of the directory where you installed the WLE software
- Make sure JAVAHOME is set to the full pathname of the directory where you installed the JDK software

For complete information on how to verify these settings, see the topic "Setting Up Your WLE Development Environment" on page 3-2.

Hello World Files

The files needed for this example are supplied on the BEA WebLogic Enterprise unsupported samples Web Site. You can get the URL for this Web site, and other related information about it, from the product Release Notes.

The files included are shown in the following table.

File	Description
`examples/hello/Hello.java`	A remote interface
`examples/hello/HelloImpl.java`	A remote object implementation that implements `examples.hello.Hello`
`examples/hello/HelloClient.java`	A client that invokes the remote method, `sayHello`
`ServerImpl.java`	Registers the RMI implementation with the WLE server at startup
`server.xml`	Server description file, which provides information about the WLE application required by the `buildjavaserver` command. When you run the `runme` script, one of the things it does is package the generated class files into a JAR file by running the WLE command `buildjavaserver` on the `server.xml` file
`runme.cmd` `runme.ksh`	Windows (DOS) and UNIX scripts, respectively, that you can run to build and run the Hello World example. The `runme` script calls on all other files listed here, and generates new files.
`clobber.cmd` `clobber.ksh`	Windows (DOS) and UNIX scripts, respectively, that you can run to remove files generated by the Hello World example.

Building and Running the Hello World Example

We suggest that first you just find the Hello World RMI example (on the Web site indicated in the product Release Notes), build it, and run it. This is an easy way to get familiar with WebLogic RMI on IIOP.

To build and run the Hello World example do the following:

1. Make sure WLE is installed on your local system, and that the following environment variables are set to indicate the appropriate paths:

   JAVA_HOME - Set to the full pathname of your Java Development Kit (JDK)

   TUXDIR - Set to the full pathname of your WLE installation directory

   The Hello World example script automatically sets the TUXCONFIG environment variable for you, so you do not need to set this variable now. The TUXCONFIG variable indicates the location of the `TUXCONFIG` file for the WLE application you want to run, in this case our Hello World example application. The script also adds the HelloWorld application classes to your CLASSPATH, and the required paths for TUXDIR and the JDK bin to your PATH.

2. Copy the RMI `helloworld` directory and files from the WLE Unsupported Samples Web page onto your local system.

   The URL for the unsupported samples WLE Web site is specified in the product Release Notes under “About This BEA WLE Release” in the subsection “Unsupported WLE Samples and Tools on BEA Web Site.” On the samples Web page, the RMI HelloWorld example is in a directory similar to the following:

   ```
   /unsupported/samples/rmi/helloworld
   ```

3. Change directories (`cd`) to your local WLE RMI `helloworld` example and type the following at the command line prompt:

   ```
   runme
   ```

Running this script compiles, builds, and runs the RMI Hello World example. You should see output similar to the following, as a result of running the `runme` script.

```
C:\myWLEapps\rmi\helloworld>runme
Setting up for RMI HelloWorld sample.
--- Verifying some variables...
--- Creating setenv.cmd...
--- Creating ubbconfig...
--- Creating run_client.cmd...
--- Compiling Java sources...
--- Generating Stub and Skeleton...
--- Building the Jar...
--- Creating tuxconfig...
--- Booting WLE...
Booting all admin and server processes in
C:\myWLEapps\rmi\helloworld\tuxconfig
INFO: BEA Engine, Version 2.4
INFO: Serial #: 123456789, Expiration 2000-06-21, Maxusers 200
INFO: Licensed to: Samantha Stevens

Booting admin processes ...

exec BBL -A :
        process id=271 ... Started.

Booting server processes ...

exec TMSYSEVT -A :
        process id=239 ... Started.
exec TMFFNAME -A -- -N -M :
        process id=240 ... Started.
exec TMFFNAME -A -- -N :

        process id=243 ... Started.
exec TMFFNAME -A -- -F :
        process id=284 ... Started.
exec JavaServer -A :
        process id=225 ... Started.
exec ISL -A -- -n //SAMS:2468 :
        process id=274 ... Started.
7 processes started.
--- Running the RMI Client... (Should say "Hello World!")...
Hello World!
--- Shutting down WLE...
Shutting down all admin and server processes in C:\myWLEapps\rmi\helloworld

Shutting down server processes ...
Server Id = 5 Group Id = GROUP1 Machine = simple: shutdown succeeded
Server Id = 6 Group Id = GROUP2 Machine = simple: shutdown succeeded
Server Id = 4 Group Id = GROUP1 Machine = simple: shutdown succeeded
```

```
Server Id = 3 Group Id = GROUP1 Machine = simple: shutdown succeeded
Server Id = 2 Group Id = GROUP1 Machine = simple: shutdown succeeded
Server Id = 1 Group Id = GROUP1 Machine = simple: shutdown succeeded
Shutting down admin processes ...

Server Id = 0 Group Id = simple Machine = simple: shutdown succeeded
7 processes stopped.
--- Finished.
C:\rmiHelloExample\helloworld>
```

Some of the tasks performed by the script are:

Sets up your WLE environment. This includes setting WLE environment variables, and creating a `UBBCONFIG` file based on your system name. The `UBBCONFIG` file is used to generate a `TUXCONFIG` file.

Generates a directory called `classes` (if it does not already exist) and adds the `classes` directory is in your local CLASSPATH (if it is not already included).

Runs the `javac` compiler on the `examples/hello/*.java` files to generate executable Java class files, and puts the generated class files under a directory called `classes`

Runs the command `java weblogic.rmic` on the implementation class `HelloImpl.class` file to generate an RMI client stub and RMI server skeleton

Packages the class files into a JAR file by running the `buildjavaserver` command on `server.xml`

Boots the WLE server (`tmboot -y`)

Runs the RMI client

Stops the WLE server (`tmshutdown -y`)

Notice also that as a result of running the `runme` script, you get several new files. Some of the more interesting ones are shown in the following table.

Generated File(s)	Description
Java class files in `classes/examples/helloworld/`	The classes `Hello.class`, `HelloClient.class`, and `HelloImpl.class` were created by running the `javac` command on `Hello.java`, `HelloClient.java`, and `HelloImpl.java`, respectively.

Generated File(s)	Description
RMI stub and skeleton classes in `classes/examples/helloworld/`	`Hello_WLStub.class` is a proxy for the client and `Hello_WLSkel.class` is a proxy class for the server. These class files were created by running the command `java weblogic.rmic` on the fully qualified package name of the implementation class, `HelloImpl.class` (java `weblogic.rmic examples.hello.HelloImpl`)
`classes/ServerImpl.class`	Registers the application at startup. This was created by running the javac command on `ServerImpl.java`.
`server.jar`	The Hello World application packaged into a Java Archive (JAR) file for deployment. This was created by running the `buildjavaserver` command on the file `server.xml`
`server.ser`	Serialized version of the server-implementation as specified in the `server-descriptor-name` section of the `server.xml` file. running the `buildjavaserver` command on the file `server.xml`
`UBBCONFIG file`	ASCII version of the WLE application configuration file containing parameters that the WLE software interprets to create an executable application.
`TUXCONFIG file`	Binary version of the WLE application configuration file. This was generated by running `tmloadcf` on the `UBBCONFIG` file.
`setenv.cmd` `setenv.ksh`	Windows (DOS) and UNIX commands to set the WLE specific environment variables APPDIR and TUXDIR based on your current environment.
`run_client.cmd` `run_client.ksh`	Windows (DOS) and UNIX commands to run the client with appropriate arguments.

For more information about these files, refer to Chapter 3, “Developing RMI Applications in WLE.”

Cleaning up the Directory

If you want to start over, you can quickly remove all generated files from the example directory by running the following command in the `helloworld` directory:

```
clobber
```

Running the `clobber` command removes all generated files for the Hello World example, leaving only the original exampe files: the Java source files, `server.xml` file, and the `runme` commands for Windows and UNIX.

Understanding the Hello World Example

After you successfully run the the RMI Hello World example, you can walk through the process used to create it by referring to Chapter 3, "Developing RMI Applications in WLE." This topic steps through the entire development and runtime process using the Hello World files as an example.

Each of the key Java source code files is explained in detail in the following subtopics:

- `Hello.java` is explained in Step 2. Write the source code for a remote interface.
- `HelloImpl.java` is explained in Step 3. Write the source code for a remote object that implements the remote interface.
- `HelloClient.java` is explained in Step 4. Write the source code for a client that invokes methods on the remote object.

This topic also explains how to compile the Java source files with the `javac` compiler, how to generate stubs and skeletons with the WebLogic RMI compiler, package the class files into a WLE application, and build and run the application in the WLE environment.

3 Developing RMI Applications in WLE

You can write your own WebLogic Enterprise (WLE) RMI classes and test them in a running WLE application by following the basic guidelines described here. We cover all the steps you need to develop a WLE RMI application from scratch. Various aspects of the Hello World example illustrate the major steps in the development process.

This topic includes the following sections:

- Setting Up Your WLE Development Environment
- Developing New RMI Classes for a WLE Application
- Building Your RMI Application in the WLE Environment
- Running Your WLE RMI Application
- Stopping the WLE Server
- Using a Script as a Shortcut for Compile and Build Steps
- Deploying Your Application

Setting Up Your WLE Development Environment

Once you have installed the WLE software and the JDK software, you need to make sure that your development environment is properly configured.

Before attempting to compile and build any WLE application, you need to ensure that certain environment variables are set on your system. In most cases, the environment variables TUXDIR and JAVA_HOME are set as part of the WLE installation procedure, and if you are running WLE sample applications the `runme` scripts typically set the others for you. However, you need to check all of these environment variables to ensure they reflect correct information and modify them whenever necessary.

Table 3-1 Setting Environment Variables for WLE Applications

Environment Variable	Description

TUXDIR	The directory path where you installed the WLE software. For example:
	Windows NT
	`set TUXDIR=c:\WLEdir`
	UNIX
	`export TUXDIR=/usr/local/WLEdir`

JAVA_HOME	The directory path where you installed the JDK software. For example:
	Windows NT
	`set JAVA_HOME=c:\JDK1.2`
	UNIX
	`export JAVA_HOME=/usr/local/JDK1.2`

Environment Variable	Description

CLASSPATH	The CLASSPATH must include the pathnames defined in TUXDIR and JAVA_HOME along with pathnames to other WLE classes. The CLASSPATH must also include the pathname of the classes for the application you are developing.
	For example:
	Windows NT
	`set WLECP=%TUXDIR%\udataobj\java\jdk`
	`set CLASSPATH=%CLASSPATH%;%WLECP%\m3.jar;%WLECP%\weblogicaux.jar`
	UNIX
	`set WLECP=${TUXDIR}/udataobj/java/jdk`
	`set CLASSPATH=${CLASSPATH}:${WLECP}/m3.jar:${WLECP}/weblogicaux.jar`
	During development, or any time you are using BEA tools, you should also set up the locale for error messages from the tools:
	`set CLASSPATH=%CLASSPATH%;%TUXDIR%\locale\java\M3` on Windows NT
	`export CLASSPATH=${CLASSPATH}:${TUXDIR}/locale/java/M3` on UNIX

PATH	The PATH must include the pathnames to the necessary bins and other directories containing executable commands. For example:
	Windows NT
	`set PATH=%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;%JAVA_HOME%\jre\bin\classic;%PATH%`
	`set PATH=%PATH%;%TUXDIR%\bin`
	UNIX
	`export PATH=${JAVA_HOME}/bin:${JAVA_HOME}/jre/bin:${JAVA_HOME}/jre/bin/classic:${PATH}`
	`export PATH=${PATH}:${TUXDIR}/bin`

Verifying/Setting Environment Variables on Windows NT

To verify on a Windows system that the information for the environment variables defined during installation is correct, do the following:

1. From the Start menu, select Settings.
2. From the Settings menu, select the Control Panel.

The Control Panel appears.

3. Click the System icon.

   The System Properties window appears.

4. Click the Environment tab.

   The Environment page appears.

5. Check the settings for TUXDIR and JAVA_HOME.

To change the settings, perform the following steps:

1. On the Environment page in the System Properties window, click the environment variable you want to change or enter the name of the environment variable in the Variable field.
2. Enter the correct information for the environment variable in the Value field.
3. Click OK to save the changes.

Verifying/Setting Environment Variables on UNIX

To verify on a UNIX system that the information for the environment variables defined during installation is correct, type the following commands at the prompt:

```
printenv <ENVIRONMENT_VARIABLE>
```

To change the settings, type the following commands at the prompt:

```
export <ENVIRONMENT_VARIABLE>=<DirectoryPath>
```

Developing New RMI Classes for a WLE Application

This section describes the steps involved in writing the source code for RMI classes, using the Java source files from the WLE RMI Hello World as code examples. We explain what characterizes an RMI application in WLE, and what elements you need to include for it to work.

This section includes step-by-step instructions on how to write RMI classes, compile the source files, generate the needed stubs and skeletons, and deploy the class files in a WLE runtime environment. The steps are:

- Step 1. Decide on package names and create directories for the source code that reflects the package names.
- Step 2. Write the source code for a remote interface. (see `Hello.java`)
- Step 3. Write the source code for a remote object that implements the remote interface. (see `HelloImpl.java`)
- Step 4. Write the source code for a client that invokes methods on the remote object. (see `HelloClient.java`)
- Step 5. Compile the source code files to create the executable RMI classes.
- Step 6. Run the WebLogic RMI compiler on the implementation class to generate stubs and skeletons.

Step 1. Decide on package names and create directories for the source code that reflects the package names.

The Java programming language requires a mapping between the fully-qualified package name of a class and the directory path to that class, so you should decide on package and directory names before you begin writing any Java code.

This mapping allows the compiler to know the directory in which to find the class files mentioned in a program. For the WLE RMI Hello World example, the package name is `examples.hello` and the Java source directory is `examples/hello`.

Step 2. Write the source code for a remote interface.

A remote object is an instance of a class that implements a *remote interface*. In WLE, a remote interface must extend the interface `java.rmi.Remote`. The `rmi.Remote` interface itself contains no method signatures—it simply acts as a tag to identify remote classes.

The interface that *you* write (extending on `rmi.Remote`) should include method signatures that will be implemented in every remote class that implements it.

Your Remote interface should have the following characteristics:

- It must be public. Otherwise a client will get an error when attempting to load a remote object that implements it.
- It must extend either `java.rmi.Remote` or `weblogic.rmi.Remote`.
- Each method must declare `java.rmi.RemoteException` or `weblogic.rmi.RemoteException` (or a superclass of `RemoteException`) in its throws clause, in addition to any application-specific exceptions.
- The data type of any remote object that is passed as an argument or return value (either directly or embedded within a local object) must be declared as the remote interface type (for example, `Hello`) not the implementation class (`HelloImpl`).

Note that these requirements are consistent with the Sun JavaSoft RMI model.

Listing 3-1 shows the Remote interface `examples.hello.Hello` from our Hello World example. The interface has only one method, `sayHello`, which returns a string to the caller.

Listing 3-1 Hello.java — A RemoteInterface

```
/*
 * Copyright (c) 2000 BEA Systems, Inc.   All Rights Reserved
 */

package examples.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * This class illustrates an interface for RMI communication.
 * @author Copyright (c) 2000 by BEA Systems Inc. All Rights Reserved.
 */
public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Step 3. Write the source code for a remote object that implements the remote interface.

A *remote object* is an instance of a class that implements a remote interface.

Now write the class that can be invoked remotely. The class should implement the remote interface you wrote in Step 2. The remote object is sometimes referred to as an RMI "server."

For example, in the source file `examples/hello/HelloImpl.java` from the RMI Hello World example we do the following:

1. Define a class (`HelloImpl`) that can be invoked remotely using the methods declared in our Remote interface.
2. Create an instance of that class (the remote object) in a `main` method. At this point, we bind the instance to a name via the Java Naming and Directory Interface (JNDI). As such, the `HelloImpl` class is the remote object that *implements* the Remote interface we defined in `Hello.java` (see Listing 3-1.)

Listing 3-2 shows the remote object `examples.hello.HelloImpl` from our Hello World example.

Listing 3-2 HelloImpl.java — A Remote Object Implementation

```
/*
 * Copyright (c) 2000 BEA Systems, Inc.   All Rights Reserved
 */
package examples.hello;

import java.rmi.RemoteException;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;


/**
 * This class is the sample server for RMI/HelloWorld.
 * It illustrates establishing one's self (to JNDI) as a remote object.
 * Also, it contains the trivial server method sayHello().
 *
```

```
 * @author Copyright (c) 2000 by BEA Systems Inc. All Rights Reserved.
 */
public class HelloImpl implements Hello {

  // Overhead to register one's self:

  private static InitialContext initialContext;

  private static Context getLocalInitialContext() throws NamingException {
    Hashtable env = new Hashtable();
    // No Context.PROVIDER_URL indicates native bootstrap
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.beasys.jndi.WLEInitialContextFactory");
    initialContext = new InitialContext(env);
    return initialContext;
  }

  public static void release() {
    try {
            initialContext.unbind("HelloServer");
    } catch (Exception e) {
            System.out.println("Couldn't unregister the HelloImpl object"
                               + e.getMessage());
            e.printStackTrace();
    }
  }

  public static void main(String args[]) {
    try {
      HelloImpl obj = new HelloImpl();
      // Bind this object instance to the name "HelloServer"
      getLocalInitialContext().bind("HelloServer", obj);
      System.out.println("HelloServer bound in JNDI");
    } catch (Exception e) {
      System.out.println("HelloImpl err: " + e.getMessage());
      e.printStackTrace();
    }
  }

  // Method(s) that the Client might call:

  public String sayHello() {
    return "Hello World!";
  }

}
```

Defining the Remote Class

As is required for RMI, our remote object implementation class, `examples.hello.HelloImpl`, does the following:

- Declares that it implements at least one remote interface. Here is the class declaration specifying that implements the interface `Hello`:

  ```
  public class HelloImpl implements Hello {
  ```

- Provides the implementation for the methods that can be invoked remotely. Here is the implementation for the `sayHello` method, which returns the string `"Hello World!"` to the caller:

  ```
  public String sayHello() {
      return "Hello World!";
  }
  ```

Creating an Instance of the Remote Class

In our example, we create the instance of the remote class (the actual remote object) in a main method as a part of our implementation class, `examples.hello.HelloImpl`.

This is fine—the class that contains the main method and instantiates the remote class can be the implementation class itself. Or, you can have the code that instantiates the remote class in another class entirely.

In the main method we do the following:

- Create an instance of a remote object `HelloImpl`:

  ```
  HelloImpl obj = new HelloImpl();
  ```

- Bind this object instance to the name "HelloServer" using JNDI `javax.naming`.

  ```
  getLocalInitialContext().bind("HelloServer", obj);
  ```

  Note that objects within WLE should be well-behaved to make administration easy. So, for every `bind` method there should be a corresponding `unbind` method somewhere. Typically, these methods are called when the server is starting (`initialize`) and stopping (`release`) as shown in Listing 3-4.

  Not doing the `unbind()` will allow clients to find the object in JNDI but get an error when they cannot use it. When the object is unavailable, it should not be listed in JNDI.

Step 4. Write the source code for a client that invokes methods on the remote object.

Finally, write a client that invokes methods on the remote object (RMI server). Listing 3-3 shows the client `examples.hello.HelloClient` from our Hello World Example.

Listing 3-3 HelloClient.java — A Client That Uses a Remote Service

```
/*
 * Copyright (c) 2000 BEA Systems, Inc.   All Rights Reserved
 */

package examples.hello;

import java.rmi.RemoteException;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;

/**
 * This class is the sample client for RMI/HelloWorld.
 * It illustrates JNDI lookup to find and use a remote object.
 *
 * @author Copyright (c) 2000 by BEA Systems Inc. All Rights Reserved.
 */
public class HelloClient {
  private static void usage() {
    System.out.println("Usage: java examples.hello.HelloClient corbaloc://<host>:<port>");
    System.exit(1);
  }

  private static Context getContext(String url) throws NamingException {
    Hashtable env = new Hashtable();
    env.put(Context.PROVIDER_URL, url);
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.beasys.jndi.WLEInitialContextFactory");
    return new InitialContext(env);
  }

  public static void main(String[] argv) throws Exception {
```

```
    if (argv.length < 1) usage();
    String url = argv[0];
    Object o = getContext(url).lookup("HelloServer");
    Hello obj = (Hello) PortableRemoteObject.narrow(o, Hello.class);
    System.out.println(obj.sayHello());
  }
}
```

Here is what `HelloClient` is doing:

- First, the client uses JNDI to get a reference to the remote object implementation (advertised as HelloServer)

  ```
  Object o = getContext(url).lookup("HelloServer");
  ```

- Once the object reference is obtained, the client *narrows* it to the appropriate type:

  ```
  Hello obj = (Hello) PortableRemoteObject.narrow(o, Hello.class);
  ```

- Finally, the client invokes the `sayHello` method on the remote object, using it in a `System.out.println` method to display the message “Hello World” on the screen:

```
System.out.println(obj.sayHello());
```

A Note about Type Narrowing

Once an object reference is obtained, the client must *narrow* it to the appropriate type. Notice the use of `PortableRemoteObject.narrow` in the following line from Listing 3-3:

```
Hello obj = (Hello) PortableRemoteObject.narrow(o, Hello.class);
```

You *could* using the cast operator here as well. However, we recommend the use of `PortableRemoteObject.narrow` to ensure interoperability with EJB container compliant implementations.

A client program that is intended to be interoperable with all compliant EJB Container implementations must use the method `javax.rmi.PortableRemoteObject.narrow` to perform type-narrowing of the client-side representations of the home and remote interface.

Programs using the cast operator for narrowing the remote and home interfaces are likely to fail if the Container implementation uses RMI-IIOP as the underlying communication transport.

Step 5. Compile the source code files to create the executable RMI classes.

We suggest that you create a separate "deployment" directory to contain the generated class files. For example, you could create a directory called *<MyWLEApps>*/rmi/helloworld/classes. You must create a deployment directory before you run the javac compiler on your source files; the javac command will not create this directory for you. (Note that in Hello World example, the runme script does create the classes directory for you before it runs the javac compiler.)

Also before you attempt to compile, set your local CLASSPATH so that it includes the pathname of your deployment directory. For example, if your deployment directory is C:\MyWLEApps\rmi\helloworld\classes, then make sure this full pathname is in your local CLASSPATH. (Note that in the Hello World example, the runme script sets this for you.)

Note: The local CLASSPATH must also include the current directory (.), along all necessary WLE and JDK pathnames. For more on setting up your development environment, refer to the topic "Setting Up Your WLE Development Environment" on page 3-2.

To compile the source files, change directories (cd) to the directory that contains the package and run the javac command on the Java source files. For the RMI Hello World example, you might cd into *<MyWLEApps>*/rmi/helloworld, then run the following command which would compile the Java source files and put the resulting class files under a directory called classes:

```
javac -d classes examples/hello/*.java
```

The preceding command creates the directory examples/hello (if it does not already exist) under classes and places the generated class files in the directory classes/examples/hello.

(In our Hello World example, this step is accomplished by running the runme script. See Step 4. Set application environment variables..)

Step 6. Run the WebLogic RMI compiler on the implementation class to generate stubs and skeletons.

To create a proxy *stub* file for the client and *skeleton* file for the server, run the `weblogic.rmic` compiler on the fully-qualified package names of compiled class files that contain remote object implementations, like `my.package.MyImpl_WLstub`. The `weblogic.rmic` command takes one or more class names as an argument and produces class files of the form `MyImpl_WLStub.class` and `MyImpl_WLSkel.class`.

To generate the stub and skeleton class files for the RMI Hello World example, you would change directories (`cd`) into the `classes` directory (in our example, *<MyWLEApps>*`/samples/rmi/helloworld/classes`) and run the `weblogic.rmic` command on the generated class file `classes/examples/hello/HelloImpl.class` as follows:

```
java weblogic.rmic -d . examples.hello.HelloImpl
```

The `weblogic.rmic` command accepts any option supported by `javac`—the options are passed directly to the Java compiler. In the example, the `-d` option indicates the root directory in which to place the compiled stub and skeleton class files. So the preceding command creates the following files in the directory *<MyWLEApps>*`/rmi/helloworld/classes/examples/hello`:

```
Hello_WLStub.class
```

```
Hello_WLSkel.class
```

The generated stub class implements exactly the same set of remote interfaces as the remote object itself, and handles the necessary encoding (*marshalling*) and decoding (*unmarshaling*) of parameters sent across the network.

(In our Hello World example, this step is accomplished by running the `runme` script. See Step 4. Set application environment variables..)

More About Stubs and Skeletons in WebLogic RMI

A *proxy* is a class used by the clients of a remote object to handle the marshalling and unmarshaling of parameters across a network. In RMI, the stub and skeleton class files that are generated by the RMI compiler are *proxies* for the RMI client and RMI server objects, respectively.

In WebLogic RMI, the RMI client stub marshals the invoked method name and its arguments for the client, forwards these to the remote object, and unmarshals the returned results for the client. An RMI client stub is generated by running the WebLogic RMI (`weblogic.rmic`) compiler on the fully-qualified package names of compiled class files that contain remote object implementations, like `my.package.MyImpl_WLstub`.

The skeleton class is also generated by the WebLogic RMI compiler but the skeleton is not used in WebLogic RMI. Generally, the RMI skeleton would unmarshal the invoked method and arguments on the remote object, invoke the method on the instance of the remote object, and then marshal the results for return to the client. WebLogic Enterprise handles the unmarshaling, method invocation, and marshalling on the RMI server side using reflection. If necessary, you can discard the generated skeleton class files to save disk space because.

More About the WebLogic RMI Compiler (weblogic.rmic)

The syntax for using the WebLogic RMI compiler is as follows:

`java weblogic.rmic` *`[options] ClassName`*

The options to the `weblogic.rmic` command are shown in the following table.

Option	Description
`-help`	Prints the complete list of command line options.
`-version`	Prints version information.
`-d <dir>`	Indicates (top-level) directory for compilation.
`-notransactions`	Skip transaction context propagation
`-verbosemethods`	Instruments proxies to print debug information to `std err`.
`-descriptor` *`<example>`*	Associates or creates a descriptor for each remote class.
`-visualCafeDebugging`	Instruments proxies to support distributed debugging under VisualCafe.
`-v1.2`	Generates Java 1.2 style stubs
`-keepgenerated`	Keeps the generated `.java` files.

Option	Description
`-commentary`	Emit commentary.
`-compiler <JavaCompiler>`	Explicitly indicate which Java compiler to use. For example: `java weblogic.rmic -compiler sj examples.hello.HelloImpl`
`-g`	Compile debugging info into class file.
`-O`	Compile with optimization on.
`-debug`	Compile with debugging on.
`-nowarn`	Compile without warnings.
`-verbose`	Compile with verbose output.
`-nowrite`	Do not generate .class files.
`-deprecation`	Warn about deprecated calls.
`-normi`	Passed through to the Symantec sj compiler.
`-J<option>`	Flags passed through to java runtime.
`-classpath <path>`	Classpath to use during compilation.

The `weblogic.rmic` command also accepts any option supported by `javac`—the options are passed directly to the Java compiler.

Building Your RMI Application in the WLE Environment

This section describes how to build an RMI application in WLE. To illustrate this, we explain the commands used in the Hello World `runme` script to compile the source files and run the WebLogic RMI code generator.

We explain in more detail how to get things set up and working in the WebLogic Enterprise environment—for Hello World, most of this is also taken care of in our `runme` script. (For example, the `runme` script generates WLE configuration information and sets up some WLE environment variables)

When you are developing your own RMI classes, you might choose to compile and build manually from the command line, or you might want to use a script similar to the one we provide with the example. Here, we clarify what the manual steps would be and point out where our script accomplishes them.

The steps are:

- Step 1. Create a mechanism for bootstrapping your application.
- Step 2. Package your application into a JAR file for deployment (buildjavaserver).
- Step 4. Set application environment variables.
- Step 3. Create a UBBCONFIG file and run tmloadcf on it to get an executable TUXCONFIG file.

Step 1. Create a mechanism for bootstrapping your application.

In Java, you use a Server object to initialize and release the server application. You implement this Server object by creating a new class that derives from the `com.beasys.Tobj.Server` class and overrides the `initialize` and `release` methods. In the server application code, you can also write a public default constructor.

For example:

```
import com.beasys.Tobj.*;

/**
* Provides code to initialize and stop the server invocation.
* ServerImpl is specified in the server.xml input file
* as the name of the Server object.
*/

public class ServerImpl
    extends com.beasys.Tobj.Server {

    public boolean initialize(string[] args)
            throws com.beasys.TobjS.InitializeFailed {
    }

    public boolean release()
            throws com.beasys.TobjS.ReleaseFailed {
    }
}
```

In the Server Description File (`server.xml`), which you process with the `buildjavaserver` command, you identify the name of the Server object.

This collection of the object's implementation and data constitutes the run-time, active instance of the CORBA object.

When your Java server application starts, the server creates the Server object specified in the XML file. Then, the server invokes the `initialize` method. If the method returns true, the server application starts. If the method throws the `com.beasys.TobjS.InitializeFailed` exception, or returns false, the server application does not start.

When the server application shuts down, the server invokes the `release` method on the Server object.

Any command-line options specified in the MODULE parameter for your specific server application in the SERVERS section of the WebLogic Enterprise domain's `UBBCONFIG` file are passed to the `public boolean initialize(string[] args)` operation as `args`.

For more information about passing arguments to the server application, see the WebLogic Enterprise Administration Guide.

For examples of passing arguments to the server application, see the *Guide to the Java Sample Applications* in the WebLogic Enterprise online documentation.

Within the `initialize` method, you can include code that does the following, if applicable:

- Creates and registers RMI objects including RMI factories.
- Allocates any machine resources, for example JDBC connections.
- Initializes any global variables needed by the server application.
- Opens the databases used by the server application.
- Opens the XA resource manager.

Writing the Code That Creates and Registers an RMI Object or Factory

For most RMI server applications you want client applications to be able to locate easily this object. You need to write the code that registers the RMI objects with JNDI, which is invoked typically as the final step of the server application initialization process.

In our Hello World example we call `HelloIMpl.main()` which handles the JNDI registration.

Releasing the Server Application

When the WebLogic Enterprise system administrator enters the `tmshutdown` command, the server invokes the following operation on the Server object of each running server application in the WebLogic Enterprise domain:

```
public void release()
```

Within the `release()`operation, you may perform any application-specific cleanup tasks that are specific to the server application, such as:

- Unregistering objects managed by the server application
- Deallocating resources
- Closing any databases
- Closing an XA resource manager

Once a server application receives a request to shut down, the server application can no longer receive requests from other remote objects. This has implications on the order in which server applications should be shut down, which is an administrative task. For example, do not shut down one server process if a second server process contains an invocation in its `release()` operation to the first server process.

During server shutdown, you may want to include an invocation to unregister each of the server application's RMI objects. For example:

```
//Unregister the object
//Use a try block since the cleanup code shouldn't throw exceptions.

try {
     HelloImpl.getLocalInitialContext().unbind("HelloServer");
     }

catch (Exception e){
     System.out.println("Couldn't unregister the HelloServer object" + e.getMessage());
    e.printStackTrace();
}
```

The invocation of the `unbind` method should be one of the first actions in the `release()` implementation. The `unbind` method unregisters the server application's objects.

Listing 3-4 shows the `ServerImpl.java` file for the RMI Hello World example.

Listing 3-4 ServerImpl.java

```
/*
 * Copyright (c) 2000 BEA Systems, Inc.   All Rights Reserved
 */

import com.beasys.Tobj.Server;
import examples.hello.HelloImpl;

/**
 * This class illustrates an interface for RMI communication.
 *
 * @author Copyright (c) 2000 by BEA Systems Inc. All Rights Reserved.
 */
public class ServerImpl extends Server {

  public boolean initialize(String[] argv) {
    try {
```

```
      HelloImpl.main(null);
    } catch (Exception e) {
      return false;
    }
    return true;
  }

  public void release() {
    HelloImpl.release();
  }
}
```

Step 2. Package your application into a JAR file for deployment (buildjavaserver).

To deploy your WLE RMI application, you need to package it into a Java archive (*JAR*) file. It is this JAR file that you will call in the WLE configuration file (`UBBCONFIG`/`TUXCONFIG`) during runtime.

This section describes how to create the JAR file using a Server descriptor file. This is what we use in the Hello World example. You could also use the JAR command to assemble your application's classes into a JAR file. But, the <ARCHIVE> element of the server descriptor file provides help by simplifying the process of collecting the files.

You can create the JAR as follows:

1. Write a server descriptor file in Extensible Markup Language (XML).

   The JAR is created in the <ARCHIVE> element. The archive element must be the last element inside the <M3-SERVER> element.

   If the XML file contains instructions to create an archive, both the class specified by `server_name` and the file specified by `server_descriptor` are stored in the archive. The `server_descriptor` file is inserted in the archive manifest with the `M3-Server` tag; this insertion makes the server descriptor the entry point during server execution.

   If you do not include the archive element, the `buildjavaserver` command generates only the server descriptor and writes it in the file specified in the `server-descriptor-name` attribute of the `M3-SERVER` element.

Listing 3-5 shows the server descriptor file for our Hello World example.

Listing 3-5 server.xml

```
<?xml version = "1.0" ?>
<!--    Copyright (c) 2000 BEA Systems, Inc.
      All Rights Reserved
-->

<!DOCTYPE M3-SERVER SYSTEM "m3.dtd">

<M3-SERVER server-descriptor-name = "server.ser"
           server-implementation  = "ServerImpl" >

  <ARCHIVE name = "server.jar">
    <CLASS name="examples.hello.HelloImpl"/>
    <CLASS name="examples.hello.Hello_WLStub"/>
    <CLASS name="examples.hello.Hello"/>
  </ARCHIVE>

</M3-SERVER>
```

2. Now run the WLE command `buildjavaserver` on your server descriptor file to create the JAR file.

   Note: The deployment directory that contains your RMI classes must be in your local CLASSPATH or `buildserver.jar` command will fail.

   For example:

   `buildjavaserver` *`<MyServer>`*`.xml`

   where *`<MyServer>`*`.xml` is your server descriptor file.

   This creates the file `server.jar`

   (In our Hello World example, the `runme` script creates the JAR by running `buildjavaserver` on the file `server.xml`. See Step 4. Set application environment variables..)

   For more information about using JAR files and Java server startup in WebLogic Enterprise, see *Steps for Creating a Java Server Application* in the WebLogic Enterprise online documentation.

Step 3. Create a UBBCONFIG file and run tmloadcf on it to get an executable TUXCONFIG file.

The configuration file is the primary means of defining the configuration of WLE applications. It consists of parameters that the WLE software interprets to create an executable application.

The `UBBCONFIG` file is an ASCII version of the configuration file. The `TUXCONFIG` file is a binary version of the configuration file that you generate from the ASCII version using the `tmloadcf` command.

In our Hello World example, the `UBBCONFIG` file is generated by the `runme` script. You can create this file manually with a text editor, too. Listing 3-6 shows a sample `UBBCONFIG` file for the Hello World example.

Listing 3-6 UBBCONFIG File for Hello World

```
*RESOURCES
IPCKEY           55432
DOMAINID  Hello
MASTER    simple
MODEL     SHM
LDBAL     N
*MACHINES
DEFAULT:
          APPDIR="C:\myWLEapps\rmi\helloworld"
          TUXCONFIG="C:\myWLEapps\rmi\helloworld\tuxconfig"
          TUXDIR="d:\wledir"
          MAXWSCLIENTS=10
"SAMS"    LMID=simple
*GROUPS
GROUP1
          LMID=simple GRPNO=1 OPENINFO=NONE
GROUP2
          LMID=simple GRPNO=2 OPENINFO=NONE
*SERVERS
DEFAULT: CLOPT="-A"
TMSYSEVT  SRVGRP=GROUP1 SRVID=1
TMFFNAME  SRVGRP=GROUP1 SRVID=2  CLOPT="-A -- -N -M"
TMFFNAME  SRVGRP=GROUP1 SRVID=3  CLOPT="-A -- -N"
TMFFNAME  SRVGRP=GROUP1 SRVID=4  CLOPT="-A -- -F"
JavaServer SRVGRP=GROUP2  SRVID=6  CLOPT="-A" MODULE="C:\myWLEapps\rmi\helloworld\server.jar"
ISL       SRVGRP=GROUP1 SRVID=5  CLOPT="-A -- -n //SAMS:2468"
```

```
*SERVICES
```

After you create the UBBCONFIG file, you must run tmLoadcf on it to create the executable TUXCONFIG file as follows:

```
tmloadcf -y ubbconfig
```

In the RMI Hello World example, this is also handled in the runme script.

The TUXCONFIG file contains information used by tmboot to start the servers and initialize the bulletin board of a BEA Tuxedo system bulletin board instantiation in an orderly sequence. The tmadmin command line utility uses the configuration file (or a copy of it) in its monitoring activity. The tmshutdown command references the configuration file for information needed to shut the application down.

You can use the tmconfig command to edit many of the parameters in the executable TUXCONFIG file while your application is running.

Step 4. Set application environment variables.

Before you can run your application, you must set the following WLE environment variables specific to the application you want to run:

- APPDIR — Specifies the full pathname to the directory that contains the WLE application you want to run. In the case of Hello World, our application might reside in the server.jar file in C:/MyWLEApps/rmi/helloworld/.
- TUXCONFIG — Specifies the full pathname of the TUXCONFIG file for the application. For the Hello World example, you could set this to C:/MyWLEApps/rmi/helloworld/tuxconfig

(In our Hello World example, our runme script sets these variables. See Step 4. Set application environment variables..)

Listing 3-7 shows an example of setting WLE environment variables on a Windows NT system.

Listing 3-7 Setting WLE Application Environment Variables on Windows NT

Systems

```
set APPDIR=C:\myWLEapps\rmi\helloworld
set TUXCONFIG=C:\myWLEapps\rmi\helloworld\tuxconfig
```

Listing 3-8 shows an example of setting WLE environment variables on a UNIX system.

Listing 3-8 Setting WLE Application Environment Variables on UNIX Systems

```
export APPDIR=$HOME/myWLEapps/rmi/helloworld
export TUXCONFIG=$HOME/myWLEapps/rmi/helloworld/tuxconfig
```

Running Your WLE RMI Application

Once you have created the RMI classes and built the application, you can test it by running it as a WLE application. To run it:

1. Make sure the application specific variables APPDIR and TUXCONFIG are set. (See Step 4. Set application environment variables.)
2. Start the WLE server by typing the following at the command line:

   `tmboot -y`
3. Run your RMI client in a form similar to this:

   `java <PackageNameOfClient> <Arguments>`

   For Our Hello World example, the command to run the client is:

   `java examples.hello.HelloClient corbaloc://<MyMachineID>`

(In out Hello World example, the `runme` script boots the WLE server and runs the client for you.)

Stopping the WLE Server

Whenever you are ready to stop the WLE server, type the following at the command line:

```
tmshutdown -y
```

(In out Hello World example, the `runme` script shuts down the WLE server for you.)

Using a Script as a Shortcut for Compile and Build Steps

In our Hello World RMI example, we use `runme` scripts that contain DOS or UNIX shell commands to handle a lot of the compile, environment setup, and build tasks detailed in the previous sections. It is very likely you will want to do this as well.

For Hello World, our `runme` script is used to accomplish the following tasks:

- Runs the `javac` compiler on the `*.java` source files to generate the `*.class` files
- Runs the `weblogic.rmic` compiler on the remote class to generate a stub and skeleton. For example, the command:

```
java weblogic.rmic -d <YourClassesDirectory> examples.hello.HelloImpl
```

  runs the `weblogic.rmic` compiler on the file `examples/hello/HelloImpl.class` and puts the resulting stub and skeleton in whatever location you specify as *`<YourClassesDirectory>`*

- Packages the class files into a JAR file (in our example, `server.jar`) by running the WLE command `buildjavaserver` on the `server.xml` file
- Creates a `UBBCONFIG` file and runs `tmloadcf` on it to generate a `TUXCONFIG` file. Sets WLE application specific environment variables (APPDIR and TUXCONFIG) before booting WLE.

- Boots the WLE server
- Runs the RMI client
- Stops the WLE server.

The `runme` scripts are located in the Hello World example `helloworld` directory. You can use a text editor to view the scripts.

Deploying Your Application

To deploy a WLE application on machines other than your development system, you need to ensure that the appropriate environment variables are set on the target systems.

Deploying the Client

For systems where you want to deploy a WLE client only, make sure the following environment variables are set.

Table 3-2 Environment Variables Needed to Run a WLE Client Application

Environment Variable	Description

TUXDIR	The directory path where you installed the WLE software. For example:
	Windows NT
	`set TUXDIR=c:\WLEdir`
	UNIX
	`export TUXDIR=/usr/local/WLEdir`

Environment Variable	Description

JAVA_HOME	The directory path where you installed the JDK software. For example:
	Windows NT
	`set JAVA_HOME=c:\JDK1.2`
	UNIX
	`export JAVA_HOME=/usr/local/JDK1.2`

CLASSPATH	The CLASSPATH must include the pathnames defined in TUXDIR and JAVA_HOME along with pathnames to other WLE classes. (The CLASSPATH must also include the pathname of the classes for the application.)
	For example:
	Windows NT
	`set WLECP=%TUXDIR%\udataobj\java\jdk`
	`set CLASSPATH=%WLECP%\m3envobj.jar;%WLECP%\weblogicaux.jar;%WLECP%\wleclient.jar;%WLECP%\wlej2eecl.jar;%CLASSPATH%`
	UNIX
	`set WLECP=${TUXDIR}/udataobj/java/jdk`
	`set CLASSPATH=${WLECP}/m3envobj.jar:${WLECP}:/weblogicaux.jar:${WLECP}/wleclient.jar:${WLECP}/wlej2eecl.jar:${CLASSPATH}`

PATH	The PATH must include the pathnames to the necessary bins and other directories containing executable commands. For example:
	Windows NT
	`set PATH=%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;%JAVA_HOME%\jre\bin\classic;%PATH%`
	UNIX
	`export PATH=${JAVA_HOME}/bin:${JAVA_HOME}/jre/bin:${JAVA_HOME}/jre/bin/classic:${PATH}`

Note that the main differences between setting environment variables for a client-only deployment versus server development or deployment is that client-only run-time systems require `m3envobj.jar`, `wleclient.jar`, and `wlej2eecl.jar` and do not require the `locale/M3` tools. Also, you can run client-only run-time systems with only the JRE bin in the PATH instead of the full JDK bin.

Deploying the Server

For systems where you want to deploy a WLE server, the environment variables must be set exactly as required for development. See the section “Setting Up Your WLE Development Environment” on page 3-2

4 Using RMI with Client-Side Callbacks

This topic includes the following sections:

- Understanding Server-to-Server Communication
- Joint Client/Server Applications
- When do I need to use callbacks?
- Example of Callbacks in RMI

Understanding Server-to-Server Communication

Server-to-server communication allows WebLogic Enterprise (WLE) applications to invoke remote objects and handle invocations from those remote objects (referred to as *callback objects*). The remote objects can be either inside or outside of a WLE domain.

WebLogic Enterprise (WLE) RMI supports client-to-server, client-to-client, and *server-to-client* invocations, with *callbacks* from server-side objects to clients. Clients can be applets or full Java client applications.

Joint Client/Server Applications

In simple usage, *client applications* invoke methods on a remote object. The *server applications* implement the methods of the remote object. The remote objects in the server application live within the WLE domain that supports security and transactions. These remote objects in the server application are referred to as *WLE objects*.

Server applications can act as client applications of other server applications. Server-to-server communication allows client applications to act as server applications for requests from other client applications or from WLE server applications.

The server-to-server communication functionality is available through a *callback object*. A callback object has two purposes:

- It invokes operations on RMI objects.
- It implements the operations of an RMI object.

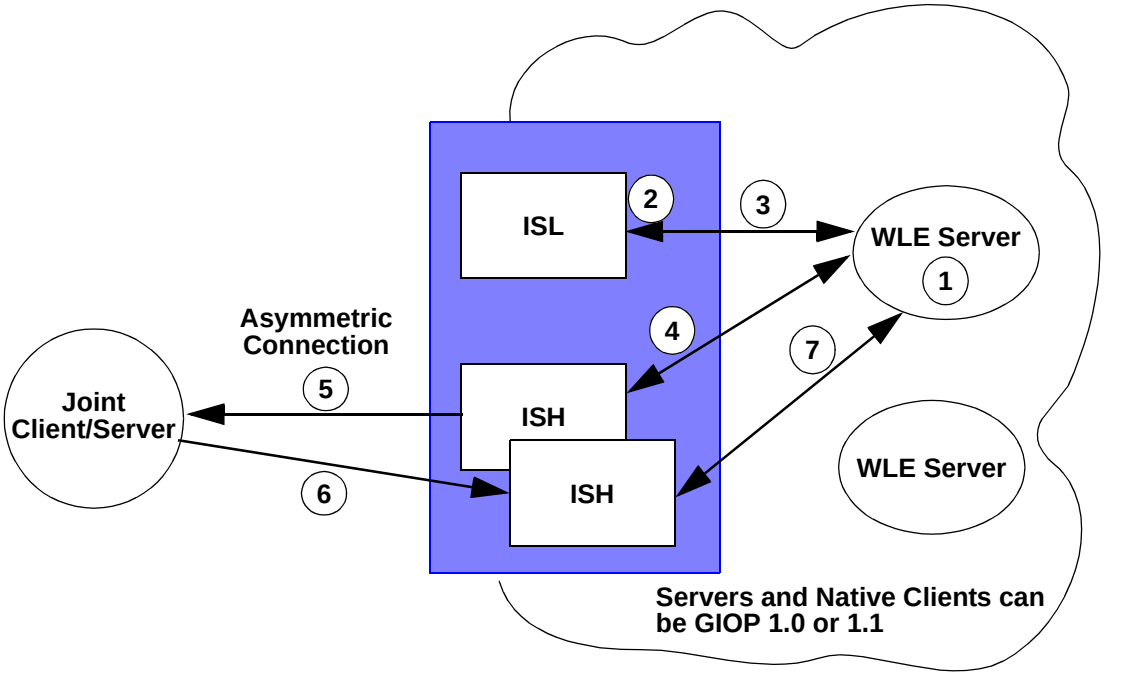
Callback objects are not subject to WLE administration, they should be used when transactional behavior, security, reliability, and scalability are not important.

Callback objects are implemented in *joint client/server applications*. A joint client/server application consists of the following:

- A portion that performs WLE client application functions, such as initializing the JNDI context, using the context to establish connections, looking up initial references to objects, and using factories to create objects.
- A portion that creates the remote object implementation (callback object) and activates the callback object.

Figure 4-1 shows the structure of a joint client/server application.

Figure 4-1 Structure of a Joint Client/Server Application



Joint client/server applications use RMI on IIOP to communicate with the WLE server in an *asymmetric* fashion. As indicated in the figure, the following operations are executed:

1. A server gets an object reference from some source. It could be a naming service or it could be passed in through a client, but not located in that client. Since the object reference is not located in a client connected to an ISH, the outgoing call cannot be made using the bidirectional method. The WLE server invokes on the object reference.
2. On the first invoke, the routing code invokes a service in the ISL and passes in the host/port.
3. The ISL selects an ISH to handle the outbound invoke and returns the ISH information to the WLE server.
4. The WLE server invokes on the ISH.

5. The ISH determines which outgoing connection to use to send the request to the client. If none is connected, the ISH creates a connection to the host/port.
6. The client executes the method and sends the reply back to the ISH.
7. The ISH receives the reply and sends it to the WLE server.

Use of callback objects in Java applets is limited due to Java applet security mechanisms. Any Java applet run-time environment that allows a Java applet to create and listen on sockets (via the proprietary environment or protocol of the Java applet) can act as a joint client/server application. However, if the Java applet run-time environment restricts socket communication, the Java applet cannot act as a joint client/server application.

Joint client/server applications use RMI on IIOP to communicate with the WLE server applications which work in an asymmetric fashion, as shown in Listing 4-1. Joint client/server applications can invoke methods on any callback object, and are not restricted to invoking callback objects implemented in joint client/server applications connected to an ISH. Asymmetric IIOP forces the WLE infrastructure to search for an available ISH to handle the invocation. The ISL controlling the ISH must have been configured with the `-O` option to support callbacks

For information on the IIOP Server Listener (ISL), see the *Administration Guide* in WebLogic Enterprise online documentation.

For a more detailed description of asymmetric IIOP, see the *CORBA Java Programming Reference* in the WebLogic Enterprise online documentation.

For more information about management and configuration on remote client applications, see *Managing Remote Client Applications (WLE Systems)* in the WebLogic Enterprise online documentation.

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).

When do I need to use callbacks?

In WLE, a particularly useful feature of RMI is that you can use it to do client callbacks from Enterprise Java Bean (EJB) servers. Clients cannot advertise EJB implementations, but they can advertise RMI implementations. So if a client wants to be called back from an EJB instance, it should create an RMI object and pass the reference to the EJB instance. The EJB instance can then invoke the client back by using the RMI reference.

In practical use, being able to use a remote object as a parameter or a return value for a remotely invoked method is convenient for such things as updating the display of an applet in response to server-side events. For example, you could simply export the applet itself as a remote object that registers an interest in server-side events, and whose display changes in response to those events.

Example of Callbacks in RMI

Writing source code for RMI applications that use client-side callbacks differs from standard RMI applications in that you have to include some additional code for a *client interface*. The *remote client* must implement the client interface. Also, the remote (server) object will now include objects received from the client and method invocations on those objects.

Figure 4-2 shows the structure of an RMI application that uses client callbacks.

Figure 4-2 RMI with Client-Side Callbacks

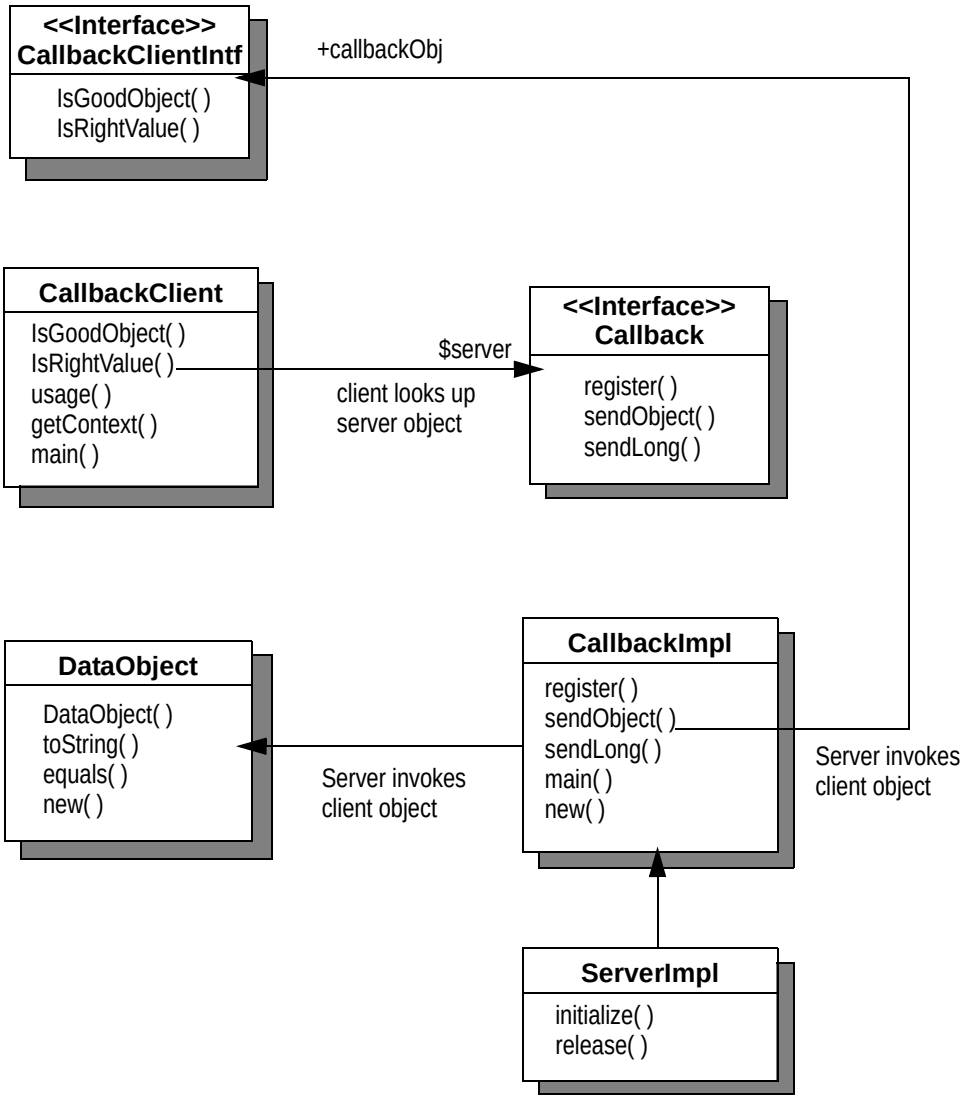
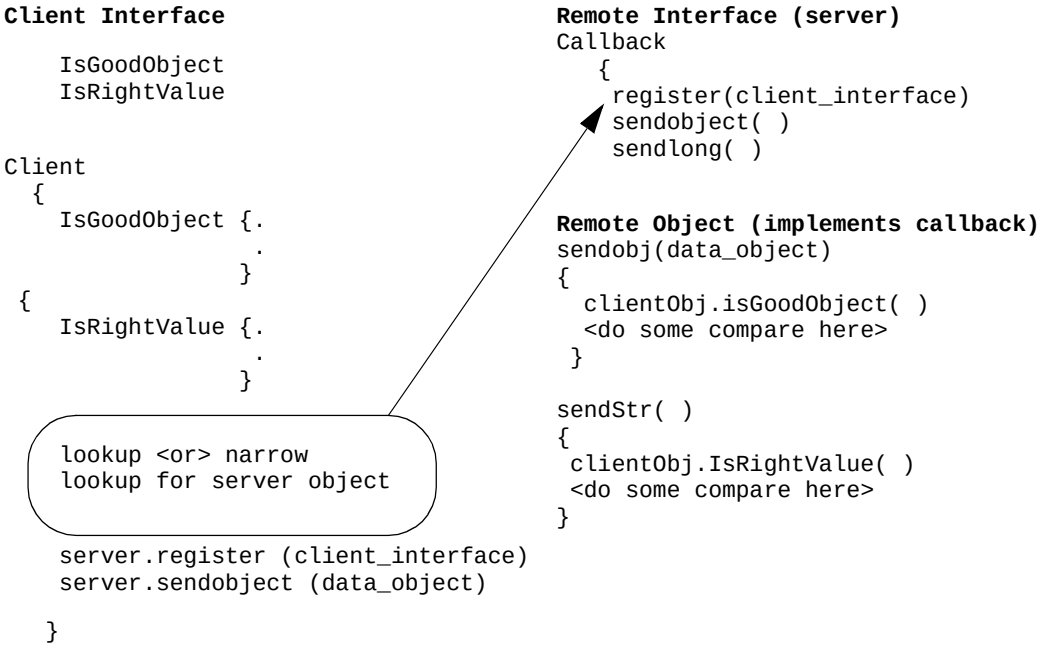


Figure 4-3 shows pseudo-code to illustrate the client-server interaction in a callback scenario.

Figure 4-3 Anatomy of RMI Client-Side Callbacks



The following sections provide a code example of a simple application that illustrates RMI client callbacks.

- The RMI Client Interface
- The RMI Client
- The RMI Remote Interface
- The Remote Object (RMI Server)
- Running the RMI Callback Example

The RMI Client Interface

Listing 4-1 shows the *client interface*. The client interface declares two business methods: `IsGoodObject` and `IsRightValue`.

Listing 4-1 CallbackClientIntf.java - A Client Interface

```
/* Copyright (c) 1999 BEA Systems, Inc.   All Rights Reserved */

import java.rmi.*;

/**
 *  CallbackClientIntf interface contains following methods
 *  IsGoodObject(Obj, Obj): compare 2 objects,
 *  IsRightValue(long, long): compare 2 longs,
 */
public interface CallbackClientIntf extends Remote
{
  public static final String NAME = "CallbackClientIntf";

  public boolean IsGoodObject(Object Obj1, Object Obj2) throws RemoteException;
  public boolean IsRightValue(long val1, long val2)
            throws RemoteException, Exception;

} // end CallbackClientIntf
```

The RMI Client

Listing 4-2 shows the RMI client implementing the client interface.

As shown in the bold code, the client does the following:

1. Implements the methods defined in the client interface, `CallbackClientInf.java`. (See all the bold code that appears *between* the star comment lines `//******`)

```
public boolean IsGoodObject(Object Obj1, Object Obj2) . .
   .
   .
public boolean IsRightValue(long val1, long val2) . .
```

```
.
.
```

2. Looks up the server object:

```
Object o = getContext(url).lookup(Callback.NAME);
server = (Callback)PortableRemoteObject.narrow(o, Callback.class);
```

3. Sends the client object to the server:

```
int s = server.register(new CallbackClient());
```

4. Invokes the business methods on the server object:

```
String errMsg = server.sendObject(new DataObject("dataobj"));
```

The `sendObject` method does a callback on the client object.

5. Invokes another business method on the server object:

```
String errMsg = server.sendLong(12345);
```

The `sendLong` method does a callback on the client object.

Listing 4-2 CallbackClient.java - A Client That Implements the Client Interface

```
/* Copyright (c) 1999 BEA Systems, Inc.   All Rights Reserved */

import java.util.Hashtable;
import java.rmi.RemoteException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;

/**
 * CallbackClient will do following steps:
 * 1. server.register(ClientObj): send a client object to server
 * 2. server.sendObject(DataObj): send a dataobject to server.
 *    server invokes ClientObj.IsGoodObject(obj, obj)
 * 3. server.sendLong(value): send a long number to server.
 *    server invokes ClientObj.IsRightValue(val, val)
 */
public class CallbackClient implements CallbackClientIntf
{
    static Callback server;  // An instance of the CallbackClientIntf

//*************************************************************
```

```
  // Implement methods of CallbackClientIntf
  // Compare 2 objects, this method is for clientobject
  public boolean IsGoodObject(Object Obj1, Object Obj2) throws RemoteException
  {
      return (Obj1.equals(Obj2));
  }

  // Compare 2 longs, this method is for clientobject
  public boolean IsRightValue(long val1, long val2) throws RemoteException
  {
      return (val1 == val2);
  }
//************************************************************

   private static void usage()
   {
      System.out.println("Usage: java CallbackClient corbaloc://<host>:<port>");
      System.exit(1);
   }

   private static Context getContext(String url) throws NamingException
   {
      Hashtable env = new Hashtable();
      env.put(Context.PROVIDER_URL, url);
      env.put(Context.INITIAL_CONTEXT_FACTORY,
               "com.beasys.jndi.WLEInitialContextFactory");
      return new InitialContext(env);
   }

   public static void main(String argv[])
   {
    if (argv.length < 1) usage();
    String url = argv[0];
    try
    {
       Object o = getContext(url).lookup(Callback.NAME);
       server = (Callback)PortableRemoteObject.narrow(o, Callback.class);
    }catch(Exception e)
    {
      System.out.println("exception in lookup server obj" + e);
    }

    //1: register ClientObject to server
     try
     {
         int s = server.register(new CallbackClient());
         if (s == Callback.FAILURE)
         {
            System.out.println("1. Couldn't send client object to server");
```

```
            System.exit(1);
         }
         else
            System.out.println("1. Success sending ClientObject to server");
     }
     catch(Exception e)
     {
         System.out.println("exception in rmiRegister: "+e);
 System.exit(1);
     }

    //2: invoke business method from server
    // send a dataobject to server
     try
     {
         String errMsg = server.sendObject(new DataObject("dataobj"));
         if (!errMsg.equals("")) {
           System.out.println("2. "+errMsg);
         } else {
           System.out.println("2. success on send data object to server");
          System.out.println("   and server callback client using ClientObject");
         }
     }
     catch(Exception e)
     {
         System.out.println("exception in sendObject(obj): "+e);
     }

    //3: invoke business method from server
    // send a string to server
     try
     {
         String errMsg = server.sendLong(12345);
         if (!errMsg.equals("")) {
           System.out.println("3. "+errMsg);
         } else {
           System.out.println("3. success on send long value to server");
          System.out.println("    and server callback client using ClientObject");
         }
     }
     catch(Exception e)
     {
        System.out.println("Exception in sendLong(value): "+e);
     }
   }

} // end CallbackClient
```

The RMI Remote Interface

Listing 4-3 shows the RMI remote interface, in which we declare the business methods:

```
public int register(Object callbackObj) throws RemoteException;
public String sendObject(Object Obj)throws RemoteException;
public String sendLong(long val) throws RemoteException, Exception;
```

Listing 4-3 Callback.java - A RMI Remote Server Interface

```
/* Copyright (c) 1999 BEA Systems, Inc.   All Rights Reserved */

import java.rmi.*;

/**
 *  Callback interface contains following methods
 *  register(callBack): send clientcallback obj to server
 *  sendObject(Obj): send an object to server
 *  sendLong(Val): send a long value to server
 */
public interface Callback extends Remote
{
  public static final String NAME = "Callback";
  public static final int FAILURE = -1;
  public static final int SUCCESS = 0;

  public int register(Object callbackObj) throws RemoteException;
  public String sendObject(Object Obj)throws RemoteException;
  public String sendLong(long val) throws RemoteException, Exception;

} // end Callback
```

The Remote Object (RMI Server)

Listing 4-4 shows the remote object implementation of the business methods in the RMI remote interface.

Listing 4-4 CallbackImpl.java - A Remote Object that Implements the RMI Remote Interface

```
/* Copyright (c) 1999 BEA Systems, Inc.   All Rights Reserved */

import java.util.Hashtable;
import java.rmi.*;
import java.rmi.server.*;
import javax.naming.*;

/**
 * Implements the methods defined in the Callback remote interface.
 */
public class CallbackImpl implements Callback
{
    private Object callbackObj;  // Object on client to verify parameters.

  // remember clientobject sent to server
    public int register(Object callbackObj) // throws RemoteException
  {
    if (callbackObj == null) return Callback.FAILURE;
    this.callbackObj = callbackObj;
    return Callback.SUCCESS;
  }

  // send regular dataobject to server
  // This method returns empty string on success or else error message.
  public String sendObject(Object Obj) throws RemoteException
  {
     // client call_back
     Object tmpObj = new DataObject("dataobj");
     if (!(callbackObj instanceof CallbackClientIntf))
      return "ClientObject is not instance of CallbackClientIntf at server side";

     // client call_back object
     if (((CallbackClientIntf)callbackObj).IsGoodObject(Obj, tmpObj))
        return "";
     else
        return "fail on send dataobject to server";
  }

  // send native type long to server
  // This method returns empty string on success or else error message.
  public String sendLong(long val) throws RemoteException, Exception
  {
     // client call_back
     if (!(callbackObj instanceof CallbackClientIntf))
      return "ClientObject is not instance of CallbackClientIntf at server side";
```

```
    // client call_back object
    if (((CallbackClientIntf)callbackObj).IsRightValue(val, 12345))
       return "";
    else
       return "fail on send long value to server";
  }

 /**
  * The main() method creates an instance of CallbackImpl class
  * and invokes the rebind() method of JNDI to register the
  * new objects.  It registers the objects with the name Callback
  * and also inform you that it has successfully completed
  * the registration process.
  */
  public static void main(String args[])
  {
      try{
          Hashtable env = new Hashtable();
          env.put(Context.PROVIDER_URL,"");
          env.put(Context.INITIAL_CONTEXT_FACTORY,
           "com.beasys.jndi.WLEInitialContextFactory");
          Context ctx = new InitialContext(env);
          ctx.rebind(Callback.NAME, new CallbackImpl());

          System.out.println("CallbackImpl created and bound in JNDI to name "
     + Callback.NAME);
       }
       catch (Exception e)
       {
          System.out.println("caught exception:"+e);
       }
  }//end main()
} // end CallbackImpl
```

Running the RMI Callback Example

If you would like to run the callback example, do the following:

1. Make sure that you development environment is properly configured for compiling and running the example, as explained in the topic “Setting Up Your WLE Development Environment” on page 3-2.

2. Create a directory where you want to build and run the example. (For example `D:\work\rmi_callback`)

3. Cut and paste the code examples provided in the previous sections into four appropriately named Java source files:

   `CallbackClientIntf.java` (shown in Listing 4-1)

   `CallbackClient.java` (shown in Listing 4-2)

   `Callback.java` (shown in Listing 4-3)

   `CallbackImpl.java` (shown in Listing 4-4)

   Include these Java source files in your `rmi_callback` directory.

4. Refer to the topic "Extra Files Needed to Run the Callback Example" on page 4-17. Cut and paste the code for these files into appropriately named files:

   `Callback.ubb` (shown in Listing 4-5)

   `DataObject.java` (shown in Listing 4-6)

   `ServerImpl.java` (shown in Listing 4-7)

   `startup.properties` (shown in Listing 4-8)

   `TestServer.xml` (shown in Listing 4-9)

   Include these files in your `rmi_callback` directory as well. Optionally, you might also want to copy the file `cleanup.cmd` shown in Listing 4-10. This provides a convenient way to remove generated files after running the example.

5. Modify the file `Callback.ubb` (shown in Listing 4-5) so that it indicates the correct values for TUXDIR, APPDIR, and so on. To determine all the changes you need to make, look for the **`#CHANGEME`** comments in the file and simply edit those lines as needed. The code you need to modify on each line is shown in bold before a **`#CHANGEME`** comment.

6. Compile the Java source files:

   ```
   javac *.java
   ```

7. Run the WebLogic RMI compiler on `CallbackImpl.class` and `CallbackClient.class` files as follows:

   ```
   java weblogic.rmic CallbackImpl CallbackClient
   ```

8. Run the `buildjavaserver` command on the XML file:

   ```
   buildjavaserver testserver.xml
   ```

9. Set the WLE environment variables APPDIR and TUXCONFIG to indicate the location of your example application and `tuxconfig` file, respectively.

Environment Variable	Example Setting
APPDIR	For example, on Windows NT: `set APPDIR=D:\work\rmi_callback`
TUXCONFIG	For example, on Windows NT: `set TUXCONFIG=D:\work\rmi_callback\tuxconfig`

10. Generate a `tuxconfig` file based on `Callback.ubb` as follows:

```
tmloadcf -y Callback.ubb
```

11. Start the WLE server:

```
tmboot -y
```

12. Run the client:

```
java CallbackClient corbaloc://<YourMachineID>:10000
```

For example:

```
java CallbackClient corbaloc://SAMS:10000
```

If the example runs successfully, the following messages are displayed on the screen:

```
1. Success sending ClientObject to server
2. success on send data object to server
    and server callback client using ClientObject
3. success on send long value to server
    and server callback client using ClientObject
```

13. Stop the WLE server:

```
tmshutdown -y
```

14. To remove the generated files, you can use the `cleanup.cmd` provided in Listing 4-10, or a similar script.

Extra Files Needed to Run the Callback Example

The following files are provided for your convenience. You can cut and paste the code for each file into an appropriately named ASCII file, and use the files to build and run the callback example in your WLE environment. The files provided here are:

- `Callback.ubb` (shown in Listing 4-5)
- `DataObject.java` (shown in Listing 4-6)
- `ServerImpl.java` (shown in Listing 4-7)
- `startup.properties` (shown in Listing 4-8)
- `TestServer.xml` (as shown in Listing 4-9)
- `cleanup.cmd` for Windows NT systems (as shown in Listing 4-10)

Notice that this example uses a startup properties file to register RMI implementations at startup. (The Hello World example shown in Chapter 2, "Getting Started with RMI — a Hello World Example," registers the RMI implementations by means of `ServerImpl.java` in a different way.) For more information on using a startup properties file, see Appendix B, "Using a Startup Properties File,"

Listing 4-5 Callback.ubb

```
# Copyright (c) 2000 BEA Systems, Inc.   All Rights Reserved


*RESOURCES
IPCKEY        80952
MASTER        SITE1
MODEL         SHM
LDBAL         Y


*MACHINES
SAMS LMID=SITE1  #CHANGEME
            TUXDIR="d:\wledir"  #CHANGEME
            APPDIR="d:\work\rmi_callback" #CHANGEME
            TUXCONFIG="d:\work\rmi_callback\tuxconfig" #CHANGEME
            MAXWSCLIENTS=5

*GROUPS
DEFAULT:LMID=SITE1
STDGRP      GRPNO=1 OPENINFO=NONE


*SERVERS
DEFAULT:    RESTART=Y MAXGEN=5 REPLYQ=Y CLOPT="-A"
TMSYSEVT    SRVGRP=STDGRP SRVID=1 RESTART=Y
TMFFNAME    SRVGRP=STDGRP  SRVID=2
            CLOPT="-A -- -N -M"
TMFFNAME    SRVGRP=STDGRP  SRVID=3
            CLOPT="-A -- -N"
TMFFNAME    SRVGRP=STDGRP  SRVID=4
            CLOPT="-A -- -F"
ISL         SRVGRP=STDGRP   SRVID=5
            CLOPT="-A -- -O -n //SAMS:10000" #CHANGEME

JavaServer  SRVGRP=STDGRP SRVID=6
            CLOPT="-A -- -M 0 testserver.jar"


*SERVICES

*INTERFACES
```

Listing 4-6 DataObject.java

```
/* Copyright (c) 1999 BEA Systems, Inc.   All Rights Reserved */

/**
 * DataObject is to test WLE RMI client callback
 */
public class DataObject implements java.io.Serializable
{
    private String s;

    DataObject(String s)
    {
        this.s = s;
    }
    public String toString()
    {
        return s;
    }
    public boolean equals(Object Obj)
    {
       return (((DataObject)Obj).s.equals(s));
    }
}
```

Listing 4-7 ServerImpl.java

```
/* Copyright (c) 1999 BEA Systems, Inc.   All Rights Reserved */

import com.beasys.rmi.Startup;
import java.io.*;
import java.util.Properties;

/*
 * The ServerImpl class provides code to initialize and stop the server
 * invocation. ServerImpl is specified in the buildjavaserver XML input
 * file as the name of the server object.
 */
public class ServerImpl extends com.beasys.Tobj.Server {

  public boolean initialize(String[] args) {
    Properties p = new Properties();
    try {
      p.load(getClass().getResourceAsStream("startup.properties"));
    } catch (IOException ioe) {
      ioe.printStackTrace();
      return false;
    }
    try {
      Startup.main(p);
      return true;
    } catch (Exception e) {
      return false;
    }
  }

  public void release() {}
}
```

Listing 4-8 startup.properties

```
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# SYSTEM STARTUP FILES - Client callback
# ----------------------------------------------
#
# Register a startup class by giving it a virtual name and
# supplying its full pathname.
# weblogic.system.startupClass.[virtual_name]=[full_pathname]
#
# Add arguments for the startup class
# weblogic.system.startupArgs.[virtual_name]=[space separated arguments]

weblogic.system.startupClass.Callback=CallbackImpl
```

Listing 4-9 TestServer.xml

```
<?xml version = "1.0" ?>

<!--   Copyright (c) 1999 BEA Systems, Inc.
-->
<!DOCTYPE M3-SERVER SYSTEM "m3.dtd">

<M3-SERVER    server-descriptor-name = "testserver.ser"
               server-implementation = "ServerImpl" >

        <ARCHIVE name = "testserver.jar">
        <CLASS name="Callback"/>
        <CLASS name="CallbackClientIntf"/>
        <CLASS name="DataObject"/>
        <CLASS name="Callback_WLStub"/>
        <CLASS name="Callback_WLSkel"/>
        <CLASS name="CallbackClientIntf_WLStub"/>
        <CLASS name="CallbackClientIntf_WLSkel"/>
        <CLASS name="CallbackImpl"/>
        <FILE name="startup.properties" prefix=""/>
        </ARCHIVE>

</M3-SERVER>
```

Listing 4-10 Cleanup.cmd

```
rm *.class
rm *.jar
rm *.ser
rm tuxconfig
rm stderr
rm stdout
rm tmsysevt.dat
rm -rf .adm
```

CHAPTER

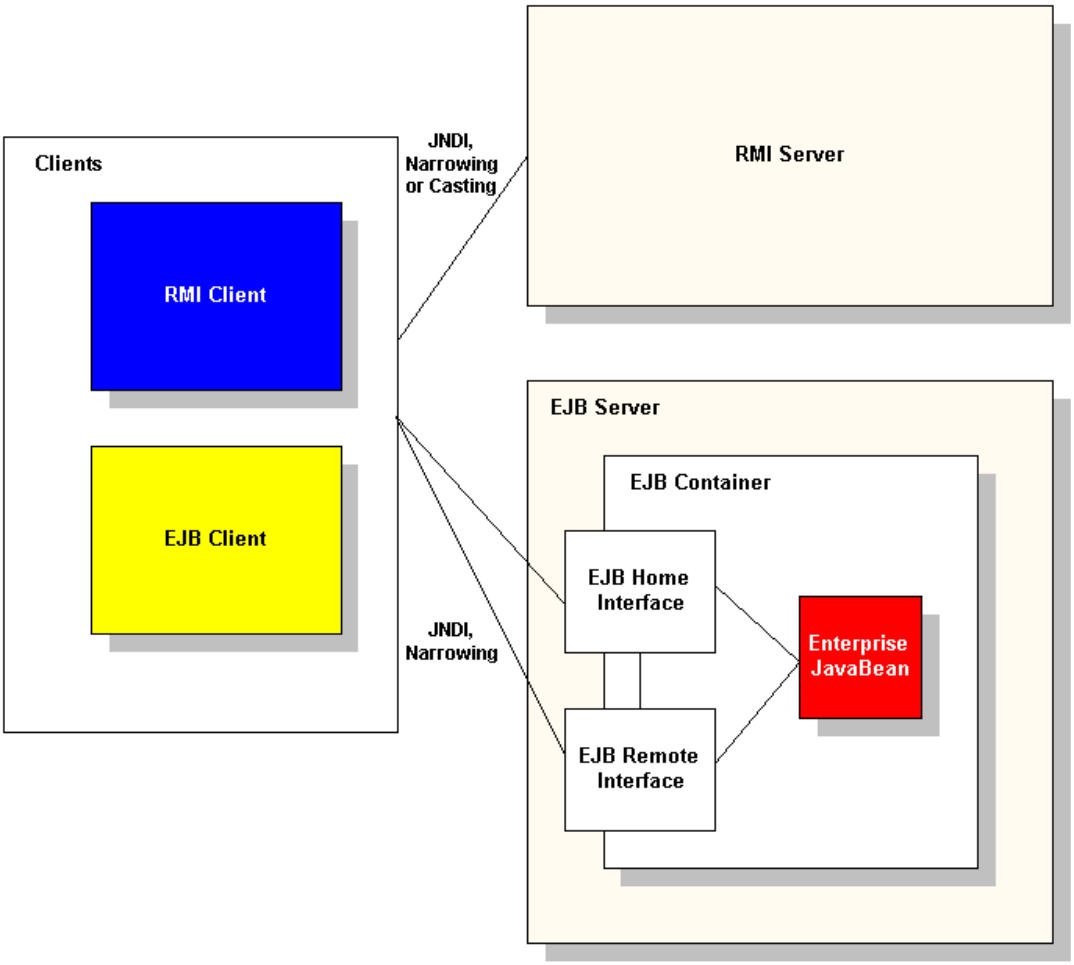
5 Using RMI with EJBs

All clients of EJBs use RMI on IIOP. There is really no difference in creating an RMI client of a traditional RMI server or of an EJB server. (See Figure 5-1 and the topic "A Note About Type Narrowing" on page 5-4.) The difference is in the way the servers handle the calls. This topic explains the relationship between WebLogic Enterprise RMI on IIOP and EJBs.

This topic includes the following sections:

- EJBs and Clients of EJBs
- Client Callbacks from EJBs
- Clients of EJBs and WLE RMI Servers
- A Note About Type Narrowing
- Where can I find examples of clients of EJBs?

Figure 5-1 All Clients of EJB Servers Use RMI on IIOP - RMI Clients and Clients of EJBs Are Essentially the Same



EJBs and Clients of EJBs

To talk to an EJB server, a client of an EJB must first obtain an object reference for the EJB server. This is the same task as an RMI client obtaining a reference to a remote object. The EJB server is always treated as a remote object. To obtain the object reference, WebLogic Enterprise clients of EJBs use the Java Naming and Directory Interface (JNDI). The JNDI call returns a reference to an object that can implement the EJB server's Home interface. The client can use the Home interface on an EJB server to look up existing EJB instances or create new ones. The client uses the Remote interface to interact with EJB objects on the server.

In short, an the client of an EJB is an RMI client that is talking to an EJB.

Client Callbacks from EJBs

In WLE, a particularly useful feature of RMI is that you can use it to do client callbacks from Enterprise Java Bean (EJB) servers. Clients cannot advertise EJB implementations, but they can advertise RMI implementations. So if a client wants to be called back from an EJB instance, it should create an RMI object and pass the reference to the EJB instance. The EJB instance can then invoke the client back by using the RMI reference.

Clients of EJBs and WLE RMI Servers

Notice that by definition all clients of EJBs use RMI on IIOP. Therefore, clients of EJBs can also talk to WLE RMI servers.

A Note About Type Narrowing

A client program that is intended to be interoperable with all compliant EJB Container implementations must use the method `javax.rmi.PortableRemoteObject.narrow` to perform type-narrowing of the client-side representations of the home and remote interface. Once an object reference is obtained, the client must *narrow* it to the appropriate type. If you are creating a vanilla RMI client, you *could* use the cast operator instead of `PortableRemoteObject.narrow`. However, we recommend the use of `PortableRemoteObject.narrow` to ensure interoperability with EJB container compliant implementations.

Where can I find examples of clients of EJBs?

All clients of EJBs use RMI on IIOP. For a description of the EJB examples, see the *WebLogic Enterprise 5.0 EJB Sample Applications* in the WebLogic Enterprise online documentation.

6 Converting Sun JavaSoft RMI to WLE RMI Classes

It is easier to use WebLogic Enterprise (WLE) RMI if you have already written classes in the Sun Microsystems JavaSoft reference implementation of RMI. This section explains how to convert Sun JavaSoft RMI classes to WebLogic RMI using the WebLogic RMI Hello World application as an example.

Suppose you have an RMI Hello World example written similar to those found in the Sun JavaSoft documentation distribution. To convert these files for use with WLE, you must do the following.

- Step 1. Modify the Java source code files.
- Step 2. Compile the Java source files.
- Step 3. Run the WebLogic RMI compiler on the implementation class.
- Step 4. Build and package the application for WLE.

Step 1. Modify the Java source code files.

To convert the RMI Hello World example from Sun JavaSoft RMI to WLE RMI, you must first modify the Java source code files to adjust for the following major differences:

- For lookup and connection bootstrapping, WLE RMI uses the Java Naming and Directory Interface (JNDI) `javax.naming` instead of `java.rmi.naming`
- WLE RMI uses JNDI `javax.naming` instead of an RMI registry
- WLE RMI does not require or recommend use of an RMI security manager
- WLE RMI classes do not extend `UnicastRemoteObject`

You need to modify the following Java source code files:

- HelloImpl.java - A Remote Object Implementation
- HelloClient.java - A Client That Invokes Methods on the Remote Object

Note that the file `Hello.java`, which contains the remote interface, is exactly the same in both the Sun JavaSoft Hello World RMI example and in WebLogic Enterprise RMI. Therefore, you do not need to make any changes to this Java source file—you can use it as is. You will need to recompile it, though, along with the other Java files.

HelloImpl.java — A Remote Object Implementation

Modify this file as follows:

1. Remove the package imports statements for the following packages, which are not used in WLE RMI:

   ```
   java.rmi.Naming
   java.rmi.RMISecurityManager
   java.rmi.server.UnicastRemoteObject
   ```

2. Add package import statements for the following Java Naming and Directory Interface (JNDI) packages, which are needed by WLE RMI:

```
java.util.Hashtable
javax.naming.Context
javax.naming.InitialContext
javax.naming.NamingException
```

3. Edit the rest of the code in this file based on the WLE RMI Hello World example to use the appropriate packages and J2EE features.

   For details, see Step 3. Write the source code for a remote object that implements the remote interface. in Chapter 3, "Developing RMI Applications in WLE."

HelloClient.java — A Client That Invokes Methods on the Remote Object

The RMI client can be either an applet or a Java client similar to that shown in our WLE RMI Hello World example. To convert either type of client from Sun JavaSoft RMI to WLE RMI, you must modify the client file similar to the following to account for some basic differences:

1. Remove the following package import statement, which is not used in WLE:

   ```
   java.rmi.Naming
   ```

2. Add package import statements for the following Java Naming and Directory Interface (JNDI) packages, which are needed by WLE RMI:

   ```
   java.util.Hashtable
   javax.naming.Context
   javax.naming.InitialContext
   javax.naming.NamingException
   ```

3. Edit the rest of the code in this file to use the appropriate packages and J2EE features. In particular, this means using JNDI for the lookup and connection bootstrapping. Once you get the object reference, be sure to use `javax.rmi.PortableRemoteObject.narrow` to narrow it to the appropriate type.

For details, see the explanation of the code for the WLE Hello World RMI client in Step 4. Write the source code for a client that invokes methods on the remote object. in Chapter 3, "Developing RMI Applications in WLE."

Step 2. Compile the Java source files.

Compile the Java source files including the remote object implementation source file (HelloImpl.java), the remote interface that it extends (`Hello.java`), the RMI client (HelloClient.java or an applet file), along with any other associated Java files needed for the application.

For example, the following command compiles the Java source files in `examples/hello` and puts the resulting class files under a directory called `classes`.

```
javac -d classes examples/hello/*.java
```

For more information on using the `javac` compiler to generate WLE RMI classes, see Step 5. Compile the source code files to create the executable RMI classes. in Chapter 3, "Developing RMI Applications in WLE."

Step 3. Run the WebLogic RMI compiler on the implementation class.

To create a proxy *stub* file for the client and *skeleton* file for the server, run the `weblogic.rmic` compiler on the fully-qualified package names of compiled class files that contain remote object implementations.

For the WLE RMI Hello World Example, you would run the `weblogic.rmic` compiler on the class file `HelloImpl` as follows:

```
java weblogic.rmic -d . examples.hello.HelloImpl
```

For more information about stubs and skeletons and about using the WebLogic RMI compiler to generate them, see Step 6. Run the WebLogic RMI compiler on the implementation class to generate stubs and skeletons. in Chapter 3, “Developing RMI Applications in WLE.”

Step 4. Build and package the application for WLE.

Once you have the WLE RMI class files, all that is left to do is create a bootstrapping mechanism for your application and package the application into a JAR file. For information on how to do this, see the topic “Building Your RMI Application in the WLE Environment” on page 3-16.

CHAPTER

7 The WebLogic Enterprise RMI API

There are several packages shipped as part of WebLogic Enterprise (WLE) RMI on IIOP. The public application programming interface (API) includes the WLE implementation of the Java RMI base classes and, for compatibility, the equivalent WebLogic Server (WLS) packages (`weblogic.rmi`). The WLE implementation also includes the WebLogic RMI code generator (`weblogic.rmic`).

Writing an application that uses remote method invocation (RMI) is essentially characterized by using the RMI API.

This topic includes the following sections:

- Overview of WebLogic Enterprise RMI Packages
- Other Java Packages Related to WebLogic Enterprise RMI
- What is different in WebLogic Enterprise RMI API?

For detailed API reference information on the packages described in this topic, see the the *WebLogic Enterprise API Javadoc page* in the WebLogic Enterprise online documentation.

Overview of WebLogic Enterprise RMI Packages

You can use either the Sun Microsystems, Inc. JavaSoft RMI related packages and classes or the BEA WebLogic RMI packages and classes to create WLE RMI applications. For compatibility with BEA WebLogic Server (WLS) the `java.rmi` classes are also implemented as `weblogic.rmi` classes.

Table 7-1 shows the Sun JavaSoft and BEA WebLogic packages that make up the "WebLogic Enterprise RMI API." The packages shown are generally supported in WLE but *with some differences* which are summarized in the table. Details on how WLE RMI differs from Sun JavaSoft RMI are provided in the section "What is different in WebLogic Enterprise RMI API?" on page 7-5. Please be sure to read this section.

Table 7-1 WLE RMI Java Packages

Package	Description	Summary of Differences
java.rmi	The java.rmi package and the weblogic.rmi.package include the interface **java.rmi.Remote** which is the basic building block for all remote objects. java.rmi.Remote contains no methods—it simply functions as a "tag" to identify remote classes. You must extend this tagging interface to create your own remote interface, with method stubs that create a structure for your remote object. Then you implement your own remote interface with a remote class. Your implementation needs to be bound to a name in the Java Naming and Directory Interface (JNDI), from whence a client or server may look up the object and use it remotely. (For more about using the JNDI API, see javax.naming.) The java.rmi package also includes several exception classes that extend **java.rmi.RemoteException**. You should code to catch these RMI exceptions in your WLE applications. Methods on remote objects should throw java.rmi.RemoteException. For details on how the WLE RMI API differs from the Sun JavaSoft implementation, see the topic "What is different in WebLogic Enterprise RMI API?" on page 7-5	Some rmi classes are not effective for use in WLE. For details, see "API Differences" on page 7-6.

Table 7-1 WLE RMI Java Packages

Package	Description	Summary of Differences
javax.rmi	The `javax.rmi` package includes one class called **`javax.rmi.PortableRemoteObject`**. The method narrow method on this class can be used in combination with the WLE implementations of JNDI and RMI. All other functions in the `PortableRemoteObject` throw `UnsupportedOperationException` in WLE. Server implementation objects may either inherit from **`javax.rmi.PortableRemoteObject`** or they may implement a remote interface and then use the `exportObject` method to register themselves as a server object. Optionally, you can use **`PortableRemoteObject.narrow`** in WLE RMI applications to perform type-narrowing, instead of a cast operator. Use of **`narrow`** is strongly recommended. A client program that is intended to be interoperable with all compliant EJB Container implementations must use the method **`PortableRemoteObject.narrow`** to perform type-narrowing of the client-side representations of the home and remote interface.	None

Other Java Packages Related to WebLogic Enterprise RMI

Table 7-2 shows other J2EE packages that provide additional functionality needed to create WLE RMI classes.

Table 7-2 Other Java Packages Related to WLE RMI

Package	Description	Summary of Differences
`javax.naming`	Provides the classes and interfaces for accessing naming services. In WLE, the recommended connection bootstrap is to create an `InitialContext`. It is created with a hash table of parameters. Some of these affect the RMI implementation.	WLE supports keys from both `javax.naming.Context`, and from `weblogic.jndi.WLContext`. For details, see the topic "Connection Bootstrapping and Security Differences" on page 7-8.
`javax.transaction`	Contains three exceptions thrown by the ORB machinery during unmarshaling.	None
`java.sql`	Provides the classes and interfaces for accessing databases via Standard Query Language (SQL).	None
`javax.sql`	The `javax.sql` API is used for bean managed persistence in EJB 1.1. Explicit access to a database starts by looking up a `javax.sql.DataSource`.	None

What is different in WebLogic Enterprise RMI API?

The WebLogic Enterprise (WLE) RMI API is a subset of the Java Development Kit 2 RMI API. As such, it supports most aspects of the Java Enterprise Edition (J2EE) including use of Java Naming and Directory Interface (JNDI) and transactions services which are needed to interact with EJBs. In WLE, RMI is hosted on IIOP which means firewalls configured to support IIOP traffic will accept WebLogic RMI on IIOP messages as standard IIOP messages.

WLE RMI supports use of RMI classes in `java.rmi`, but you need to be aware of the specific implementation of these packages in the WLE RMI development environment. WLE RMI differs from the Sun JavaSoft RMI implementation. Keep these differences in mind when you are:

- Creating new RMI applications in WLE especially if you have previous experience with the BEA WebLogic Server or the JavaSoft RMI classes.
- Converting existing RMI classes from Sun JavaSoft RMI classes. (For step-by-step instructions on how to convert existing RMI classes to WLE classes refer to Chapter 6, "Converting Sun JavaSoft RMI to WLE RMI Classes.")

The differences are summarized in the following sections:

- API Differences
- Connection Bootstrapping and Security Differences
- Tool Differences
- Configuration Differences

API Differences

Table 7-3 WLE RMI API Differences

Sun JavaSoft RMI Classes	BEA WebLogic Enterprise (WLE) RMI on IIOP
Naming, Connecting, and Bootstrapping	
`java.rmi.Naming`	Use of `rmi.Naming` is not effective for use in WLE. For developing WLE RMI applications, use Java Naming and Directory Interface (JNDI) `javax.naming` instead of the RMI registry. `java.rmi.Naming` classes relate to the Sun Javasoft implementation of the RMI registry. BEA WLE provides no equivalent. `java.rmi.Naming` classes will actually compile and may run in WLE, but will produce errors and undesired results.

Table 7-3 WLE RMI API Differences

Sun JavaSoft RMI Classes	BEA WebLogic Enterprise (WLE) RMI on IIOP
`java.rmi.registry.LocateRegistry`	Use of RMI registry related classes is not effective in WLE. For developing WLE RMI applications, use JNDI `javax.naming` instead. Registry classes relate to the Sun Microsystem Javasoft implementation of the RMI registry. BEA WLE provides no equivalent. `java.rmi.registry` classes will actually compile and may run in WLE, but will produce errors and undesired results.
Security	

`java.rmi.RMISecurityManager`
`java.rmi.server.RMISocketFactory`
`java.rmi.server.RMIClassLoader`
`java.rmi.server.UnicastRemoteObject`
 	
`LoaderHandler, Operation, RemoteCall, RemoteRef, RemoteStub, Skeleton from java.rmi.server` (deprecated in 1.2 without replacement)
	Use of the RMI class `RMISecurityManager` is not effective in WLE.
	For developing WLE RMI applications, use JNDI to specify security instead.
	`RMISecurityManager` classes will actually compile and may run in WLE, but will produce errors and undesired results.

java.rmi.RemoteServer	
Static method `getClientHost` in `java.rmi.server.RemoteServer`	Use of `rmi.server` classes is not effective in WLE. `java.rmi.server` classes will actually compile and may run in WLE, but will produce errors and undesired results.
`getLog` and `setLog` in `java.rmi.server.RemoteServer`	Use of `rmi.server` classes is not effective in WLE. `java.rmi.server` classes will actually compile and may run in WLE, but will produce errors and undesired results.
Stubs and Skeletons	
`java.rmi.RemoteObject` `java.rmi.server.RemoteStub`	Use of `rmi.RemoteObject` is not effective in WLE. `java.rmi.RemoteObject` classes will actually compile and may run in WLE, but will produce errors and undesired results.

Table 7-3 WLE RMI API Differences

Sun JavaSoft RMI Classes	BEA WebLogic Enterprise (WLE) RMI on IIOP
`SkeletonMismatchException` and `SkeletonNotFoundException` in `java.rmi.server` (deprecated in JDK 1.2)	These exception classes from `java.rmi.server` will compile and run in WLE. These classes are not actually used by WLE though. WLE uses reflection instead of skeletons.
Others	
`java.rmi.dgc.Lease` and VMID (not usable)	Not supported in WLE
`java.rmi.server.LogStream` (deprecated in JDK 1.2 without replacement)	Not supported in WLE
`java.rmi.server.ObjID` (not usable)	Use of `rmi.server` classes is not effective in WLE. `java.rmi.server` classes that use `ObjID` might actually compile and run in WLE, but will produce errors and undesired results.

Connection Bootstrapping and Security Differences

In WLE RMI, connection bootstrapping is achieved by creating an `InitialContext` via the Java Naming and Directory Interface (JNDI) with `javax.naming`.

Optionally, the JNDI `WLEContext.SECURITY_AUTHENTICATION` property can be used for security. Also, the property keys shown in the topic "JNDI Property Keys for BEA Tuxedo Style Authentication" on page 7-11 can be used for BEA Tuxedo style authentication.

For more information about JNDI, see *Using the WLE SPI Implementation for JNDI* in the WebLogic Enterprise online documentation.

For more information about using JNDI for security, see *Writing a WLE Enterprise JavaBean that Implements Security* in *Using Security* in the WebLogic Enterprise online documentation.

JNDI Environment Properties

All J2EE Java remote client applications must first create environment properties. The initial context factory uses the various properties to customize the `InitialContext` for a specific environment. You can set these properties by using a **Hashtable**. These properties, which are name-to-value pairs, determine how the `WLEInitialContextFactory` creates the `WLEContext`:

WLEContext.INITIAL_CONTEXT_FACTORY

Set this property to the WLE initial context factory "`com.beasys.jndi.WLEInitialContextFactory`" to access the WLE domain and remote naming services.

The class `com.beasys.jndi.WLEInitialContextFactory` provides the implementation for delegating JNDI methods to the WLE JNDI implementation. The `com.beasys.jndi.WLEInitialContextFactory` provides an entry point for a client into the WLE domain namespace. (See Listing 7-1 for an example.)

Listing 7-1 WLEContext.INITIAL_CONTEXT_FACTORY Property Example

```
Hashtable env = new Hashtable();
/*
* Specify the initial context implementation to use.
* The service provider supplies the factory class.
*/
env.put(WLEContext.INITIAL_CONTEXT_FACTORY,
        "com.beasys.jndi.WLEInitialContextFactory");
   .
   .
   .
```

WLEContext.PROVIDER_URL

Set the URL of the service provider with the property name `java.naming.provider.url`. This property value should specify an IIOP Listener/Handler for the desired WLE target domain. (See Listing 7-2 for an example.)

Listing 7-2 WLEContext.PROVIDER_URL Property Example

```
   .
   .
   .
env.put(WLEContext.PROVIDER_URL,
       "corbaloc://myhost:1000");
   .
   .
   .
```

The host and port combination that is specified in the URL must match the ISL parameter in the WLE domain's `UBBCONFIG` file. The format of the host and port combination, as well as the capitalization, must match. If the addresses do not match, the communication with the WLE domain fails.

A WLE server that acts as a client must set the `WLEContext.PROVIDER_URL` property as an empty string or null. The server client connects to the current application in which it is booted.

WLEContext.SECURITY_AUTHENTICATION

The WLE system supports different levels of authentication. The SECURITY_AUTHENTICATION value determines whether certificate-based SSL authentication authentication is attempted or BEA TUXEDO style authentication is used.

Valid values for this property key are "`none`", "`simple`", or "`strong`", as recommended by the Sun Microsystems Inc. JNDI specification. (See Listing 7-3 for an example.)

Listing 7-3 WLEContext.SECURITY_AUTHENTICATION Example

```
   .
   .
   .
env.put(WLEContext.SECURITY_AUTHENTICATION,
        "strong");
.
```

.
.

If “none” is specified then no authentication is attempted.

If “strong” is specified then certificate-based authentication is attempted using SSL protocols.

If the SECURITY_AUTHENTICATION “simple” or not specified, then the BEA Tuxedo style authentication is used. See the next section for information about the WLE specific keys used to support BEA TUXEDO style authentication.

JNDI Property Keys for BEA Tuxedo Style Authentication

WLE supports use of the several keys from javax.naming.Context for security as shown in Table 7-4.

Table 7-4 WLE Property Keys for Security

Key	Description
WLEContext.SECURITY_PRINCIPAL	Specifies the identity of the principal used when authenticating the caller to the WLE domain.
WLEContext.SECURITY_CREDENTIALS	Specifies the credentials of the principal when authenticating the caller to the WLE domain. ■ For certificate-based authentication enabled via SECURITY_AUTHENTICATION=”strong”, it specifies the pass phrase used to access the private key and certificate for the EJB. ■ For password-based authentication enabled via SECURITY_AUTHENTICATION=”simple”, it specifies a string that is the user’s password or an arbitrary object user_data used by the authentication server (AUTHSVR) to verify the credentials of the EJB.
WLEContext.CLIENT_NAME	Specifies the name of the EJB defined by the -c option of the tpusradd command.

Table 7-4 WLE Property Keys for Security

Key	Description
`WLEContext.SYSTEM_PASSWORD`	The system password. Required only when using Username/Password authentication.

Listing 7-4 includes the WLE keys used to define Username/Password authentication.

Listing 7-4 WLE Keys for Username/Password Authentication

```
...
//Password-Based Authentication
env.put(WLEContext.SECURITY_AUTHENTICATION, "simple");
env.put(WLEContext.SYSTEM_PASSWORD, "RMI");
env.put(WLEContext.SECURITY_PRINCIPAL, "sams");
env.put(WLEContext.CLIENT_NAME, "writers");
env.put(WLEContext.SECURITY_CREDENTIALS, "password");
```

Listing includes the WLE keys used to define certificate-based authentication.

Listing 7-5 WLE Keys for Certificate-Based Authentication

```
...
//Certificate-Based Authentication
env.put(WLEContext.SECURITY_AUTHENTICATION, "strong");
env.put(WLEContext.SYSTEM_PASSWORD, "SSL");
env.put(WLEContext.SECURITY_PRINCIPAL, "sams");
env.put(WLEContext.SECURITY_CREDENTIALS, "credentials");
...
```

Tool Differences

Stubs and skeletons for WLE RMI applications are generated by running the Weblogic RMI compiler (`weblogic.rmic`) against the remote class. A stub is the client-side proxy for a remote object that forwards each WLE RMI call to its matching server-side skeleton, which in turn forwards the call to the actual remote object implementation.

WLE does not support `java.rmi.Naming` and therefore has no `rmiregistry` tool. (Use of JNDI is supported instead.)

Configuration Differences

The only RMI configuration property used for WLE RMI is `weblogic.system.startupClass.<virtualName>` which is used to register the RMI implementation at startup time. An example of using a startup properties file is provided in Appendix B, “Using a Startup Properties File.”

The Javasoft RMI specification defines several properties. None of these have any effect on the WLE RMI implementations.

A Java Server Startup

A Java server is represented by one or multiple JAR archives containing all the application class files needed for the server to execute. Multiple JARs can be specified at boot time in `UBBCONFIG` or added at runtime. The JAR file can be built either from `buildjavaserver` tool or `ejbc` tool.

The WebLogic Enterprise (WLE) Server implementation class has `initialize` and `release` methods for handling the startup and shutdown classes. As the `initialize` method of the Server implementation class is invoked with the application arguments passed in immediately after the JAR file is loaded at JavaServer startup, so any server initialization and startup functions can be performed there. The *release* method will be called when java server is shut down. The name of the startup/shutdown classes, and the startup arguments can be specified as the application arguments after the jarfile name in MODULE attribute [WLE ADMINFS] in SERVERS section of UBBCONFIG file; or in the `startup.properties` file (same as WLS) that is packaged into the JAR file.

For an example of how RMI startup and shutdown classes (specified in a properties file) are processed in `initialize` and `release` methods of the Server implementation class, see Appendix B, “Using a Startup Properties File.”

Startup/Shutdown Classes

For each Jar file, there is only one Server implementation class that has `initialize` and `release` methods. When the JavaServer boots, it will load all the JAR files specified in UBBCONFIG, and invokes the `initialize` method of the Server implementation class with the application arguments for each JAR. At JavaServer shutdown it will invoke the `release` method.

The startup/shutdown information can be specified in the M3 server descriptor XML file that will be serialized by the `buildjavaserver` command. Also, the startup/shutdown information can be specified as EJB XML deployment descriptor extensions and will be packaged into the deployable JAR by the `ejbc` tool.

If there is only one startup class, the class can be implemented as the Server implementation class, and its `initialize` method will be called when the JAR is deployed.

If there are multiple startup classes, these can still utilize the Server implementation class with the startup class names and arguments passed as the arguments to `initialize` method.

Alternatively, the startup/shutdown classes names and arguments can be specified in a separate file `startup.properties` (same format as WLS), and be processed in the `initialize` method of the Server implementation class. An example of this is provided in Appendix B, “Using a Startup Properties File.”

Jar Tool / XML

You can use the WLE `buildjavaserver` command to generate the JAR file from an XML file. For a description of the Server class and XML file syntax, see *Steps for Creating a Java Server Application* in *Creating Java Server Applications* in the WebLogic Enterprise online documentation.

Alternatively, you can use the `ejbc` tool to package the EJB deployment descriptor extensions XML file into the deployable JAR. For more information about using the ejbc tool, see the `ejbc` command in the *Commands Reference* in the WebLogic Enterprise online documentation.

UBBCONFIG

In the SERVERS section of the `UBBCONFIG`, set `MODULE="`*`jarfilename arg1 arg2 arg3…`*`"` to specify the JAR file that was generated from `buildjavaserver` and optional application-specific arguments. You can include multiple instances of MODULE for multiple JARs. The `jarfilename` can be a fully qualified path to the location of the JAR file; or it can be relative to the directory specified by the environment variable APPDIR. Note that you can specify the JAR file and arguments in CLOPT, but you must not specify JAR files in *both* CLOPT and MODULE. We recommend that you use MODULE instead of CLOPT.

B Using a Startup Properties File

This topic provides an example of how to use a startup properties file to register RMI implementations at startup. (The RMI Hello World example registers the RMI implementations by means of `ServerImpl.java` in a different way.)

Here, we show how to specify the startup file in the `server.xml` file by means of the FILE element so that `buildjavaserver` can package the properties file in the JAR. Note that the ARCHIVE element in the XML file is optional, as the JAR file can also be generated by the JAR tool as a separate step outside of `buildjavaserver`. We also provide some sample code to demonstrate how the startup classes mechanism is implemented in the `initialize` method of the Server implementation class for RMI.

This topic includes the following sections:

- XML File
- Properties File – startup.properties
- ServerImpl Class

XML File

```
<?xml version = "1.0" ?>

<!DOCTYPE M3-SERVER SYSTEM "m3.dtd">

<M3-SERVER    server-descriptor-name = "rmi.ser"
              server-implementation = "ServerImpl" >

       <ARCHIVE name = "rmi.jar">
              <CLASS name="Simp"/>
              <CLASS name="Simp_WLStub"/>
              <CLASS name="SimpImpl"/>
              <CLASS name="SimpFactory"/>
              <CLASS name="SimpFactory_WLStub"/>
              <FILE  name="startup.properties" prefix=""/>
       </ARCHIVE>
</M3-SERVER>
```

Properties File – startup.properties

```
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# SYSTEM STARTUP FILES - Examples
# ---------------------------------------------
# Register a startup class by giving it a virtual name and
# supplying its full pathname.
# weblogic.system.startupClass.[virtual_name]=[full_pathname]
#
# Add arguments for the startup class
# weblogic.system.startupArgs.[virtual_name]=[space separated
# arguments]

weblogic.system.startupClass.simp=SimpFactoryImpl
#weblogic.system.startupArgs.simp=-inproc -second
```

ServerImpl Class

```
import java.lang.reflect.*;
import java.util.*;
import weblogic.utils.StringUtils;
import com.beasys.rmi.Startup;

public class ServerImpl extends com.beasys.Tobj.Server {
   public boolean initialize(String[] argv) {
    try {
Startup.main(getClass().getResourceAsStream("startup.properties"));
    } catch (Exception e) {
      return false;
    }
    return true;
  }

  public void release() {}
}
```

Index

W