THE ENTERPRISE MIDDLEWARE SOLUTION

BEA WebLogic Enterprise

Guide to the Java Sample Applications

BEA WebLogic Enterprise 4.2
Document Edition 4.2
July 1999

Guide to the Java Sample Applications

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

Contents

Preface

Purpose of This Document

This document describes the Java sample applications that are provided with the BEA WebLogic Enterprise (sometimes referred to as WLE) software and is intended to be used with the following documents:

- *Getting Started*
- *Creating Client Applications*
- *Creating Java Server Applications*

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

Who Should Read This Document

This document is intended for application designers and client and server programmers who would find a set of progressive examples useful in understanding the WebLogic Enterprise software.

How This Document Is Organized

The *Guide to the Java Sample Applications* is organized as follows:

- Chapter 1, “Introduction,” provides an overview of the sample applications.

- Chapter 2, “The Java Simpapp Sample Application,” describes how to build and use the Java Simpapp sample application.
- Chapter 3, “The JDBC Bankapp Sample Application,” describes how to build and use the JDBC Bankapp sample application.
- Chapter 4, “The XA Bankapp Sample Application,” describes how to build and use the XA Bankapp sample application.

How to Use This Document

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

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

Opening the Document in a Web Browser

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

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

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

Printing from a Web Browser

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

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

Documentation Conventions

The following documentation conventions are used throughout this document.

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

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

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*

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

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

`|`	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` *`firstfile-syntax`*`] [-l` *`lastfile-syntax`*`]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Related Documentation

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

BEA WebLogic Enterprise Documentation

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

Installation Guide

C++ Release Notes

Java Release Notes

Getting Started

Guide to the University Sample Applications

Guide to the Java Sample Applications (this document)

Creating Client Applications

Creating C++ Server Applications

Creating Java Server Applications

Administration Guide

Using Server-to-Server Communication

C++ Programming Reference

Java Programming Reference

Java API Reference

JDBC Driver Programming Reference

System Messages

Glossary

Technical Articles

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

BEA Publications

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

To access these documents:

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

Other Publications

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Contact Information

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

Documentation Support

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

Customer Support

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

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

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

CHAPTER

1 Introduction

This chapter provides the following information:

- An overview of the sample applications

Overview of the Sample Applications

The sample applications provide client and server programmers with the basic concepts of developing Java server applications for the Weblogic Enterprise (WLE) software.

The following sample applications are provided:

- Java Simpapp—provides a Java client application and a Java server application. The Java server application contains two operations that manipulate strings received from the Java client application.
- JDBC Bankapp—implements an automatic teller machine (ATM) interface and uses Java Database Connectivity (JDBC) to access a database that stores account and customer information.
- XA Bankapp—implements the same ATM interface as JDBC Bankapp; however, XA Bankapp uses a database XA library to demonstrate using the Transaction Manager to coordinate transactions.

The chapters in this manual describe how to build and run the sample applications.

For a description of the development process used to create the sample applications, see *Getting Started*.

CHAPTER

2 The Java Simpapp Sample Application

The chapter discusses the following topics:

- How the Java Simpapp sample application works
- Software prerequisites
- The Object Management Group (OMG) Interface Definition Language (IDL) for the Java Simpapp sample application
- Building and running the Java Simpapp sample application
- Using the Java Simpapp sample application
- Using the C++ client application with the Java Simpapp sample application
- Stopping the Java Simpapp Sample Application

Refer to `Readme.txt` in the `\WLEdir\samples\corba\simpapp_java` directory for troubleshooting information and the latest information about using the Java Simpapp sample application.

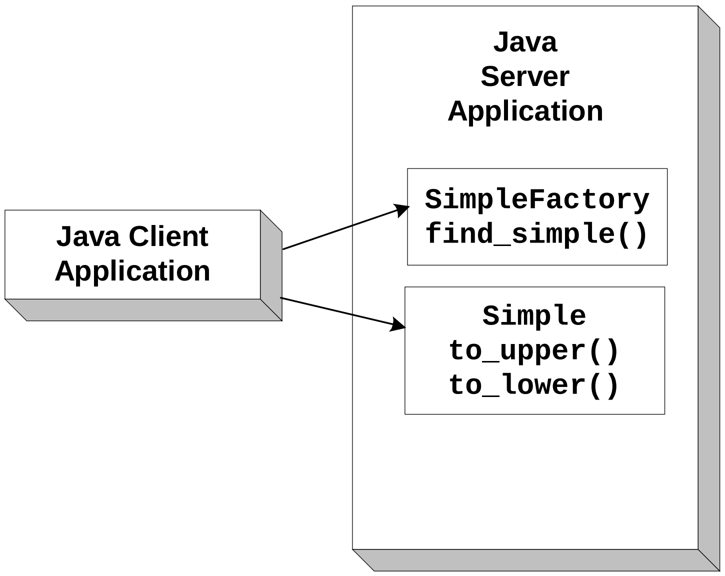
How the Java Simpapp Sample Application Works

The Java Simpapp sample application consists of a Java client application that sends requests to a Java server application. The Java server application provides an implementation of a CORBA object that has the following two methods:

- ♦ The `upper` method accepts a string from the Java client application and converts the string to uppercase letters.
- ♦ The `lower` method accepts a string from the Java client application and converts the string to lowercase letters.

Figure 2-1 illustrates how the Java Simpapp sample application works.

Figure 2-1 The Java Simpapp Sample Application



Software Prerequisites

To run the `idltojava` compiler used by the Java Simpapp sample application, you need to install Visual C++ Version 5.0 with Service Pack 3 for Visual Studio.

The OMG IDL Code for the Java Simpapp Sample Application

The Java Simpapp sample application implements the CORBA interfaces listed in Table 2-1:

Table 2-1 CORBA Interfaces for the Java Simpapp Sample Application

Interface	Description	Operation
`SimpleFactory`	Creates object references to the `Simple` object	`find_simple()`

`Simple`	Converts the case of a string	`to_upper()`
		`to_lower()`

Listing 2-1 shows the `simple.idl` file that defines the CORBA interfaces in the Java Simpapp sample application. This is the same OMG IDL file used by the C++ Simpapp sample application shipped with version 4.2 of the WebLogic Enterprise (WLE) software. The `runme` command automatically copies it from the `\corba\simpapp_java` directory.

Listing 2-1 OMG IDL Code for the Java Simpapp Sample Application

```
#pragma prefix "beasys.com"

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

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

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

Building and Running the Java Simpapp Sample Application

Perform the following steps to build and run the Java Simpapp sample application:

1. Copy the files for the Java Simpapp sample application into a work directory.
2. Change the protection attribute on the files for the Java Simpapp sample application.
3. Verify the environment variables.
4. Execute the `runme` command.

The following sections describe these steps.

Copying the Files for the Java Simpapp Sample Application into a Work Directory

You need to copy the files for the Java Simpapp sample application into a work directory on your local machine. The files for the Java Simpapp sample application are located in the following directories:

Windows NT

`drive:\WLEdir\samples\corba\simpapp_java`

UNIX

`/usr/local/WLedir/samples/corba/simapp_java`

You will use the files listed in Table 2-2 to build and run the Java Simpapp sample application.

Table 2-2 Files Included in the Java Simpapp Sample Application

File	Description
`Simple.idl`	The OMG IDL code that declares the `Simple` and `SimpleFactory` interfaces.This file is copied from the WLE `simpapp_java` directory by the `runme` command file.
`ServerImpl.java`	The Java source code that overrides the `Server.initialize` and `Server.release` methods.
`SimpleClient.java`	The Java source code for the client application in the Java Simpapp sample application.
`SimpleFactoryImpl.java`	The Java source code that implements the `SimpleFactory` methods.
`SimpleImpl.java`	The Java source code that implements the `Simple` methods.

Table 2-2 Files Included in the Java Simpapp Sample Application

File	Description
`Simple.xml`	The Server Description File used to associate activation and transaction policy values with CORBA interfaces. For the Java Simpapp sample application, the `Simple` and `SimpleFactory` interfaces have an activation policy of `method` and a transaction policy of `optional`.
`Readme.txt`	Provides the latest information about building and running the Java Simpapp sample application.
`runme.cmd`	The Windows NT batch file that builds and runs the Java Simpapp sample application.
`runme.ksh`	The UNIX Korn shell script that builds and executes the Java Simpapp sample application.
`makefile.mk`	The make file for the Java Simpapp sample application on the UNIX operating system. This file is used to manually build the Java Simpapp sample application. Refer to the `Readme.txt` file for information about manually building the Java Simpapp sample application. The UNIX `make` command needs to be in the path of your machine.
`makefiles.nt`	The make file for the Java Simpapp sample application on the Windows NT operating system. This make file can be used directly by the Visual C++ `nmake` command. This file is used to manually build the Java Simpapp sample application. Refer to the `Readme.txt` file for information about manually building the Java Simpapp sample application. The Windows NT `nmake` command needs to be in the path of your machine.
`smakefile.nt`	The make file for the Java Simpapp sample application that is used with Visual Cafe `smake` command. **Note:** `makefile.nt` is included by `smakefile.nt`.

Changing the Protection Attribute on the Files for the Java Simpapp Sample Application

During the installation of the WLE software, the sample application files are marked read-only. Before you can edit or build the files in the Java Simpapp sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

Windows NT

```
prompt>attrib -r drive:\workdirectory\*.*
```

UNIX

```
prompt>/bin/ksh
```

```
ksh prompt>chmod u+w /workdirectory/*.*
```

On the UNIX operating system platform, you also need to change the permission of `runme.ksh` to give execute permission to the file, as follows:

```
ksh prompt>chmod +x runme.ksh
```

Verifying the Settings of the Environment Variables

Before building and running the Java Simpapp sample application, you need to ensure that certain environment variables are set on your system. In most cases, these environment variables are set as part of the installation procedure. However, you need to check the environment variables to ensure they reflect correct information.

Table 2-3 lists the environment variables required to run the Java Simpapp sample application.

Table 2-3 Required Environment Variables for the Java Simpapp Sample Application

Environment Variable	Description

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

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

To verify that the information for the environment variables defined during installation is correct, perform the following steps:

Windows NT

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`.

UNIX

```
ksh prompt>printenv TUXDIR
```

```
ksh prompt>printenv JAVA_HOME
```

To change the settings, perform the following steps:

Windows NT

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.

UNIX

```
ksh prompt>export TUXDIR=directorypath
```

```
ksh prompt>export JAVA_HOME=directorypath
```

Executing the runme Command

The `runme` command automates the following steps:

1. Setting the system environment variables
2. Loading the `UBBCONFIG` file
3. Compiling the code for the client application
4. Compiling the code for the server application
5. Starting the server application using the `tmboot` command
6. Starting the client application
7. Stopping the server application using the `tmshutdown` command

Note: You can also run the Java Simpapp sample application manually. The steps for manually running the Java Simpapp sample application are described in the `Readme.txt` file.

To build and run the Java Simpapp sample application, enter the `runme` command, as follows:

Windows NT

```
prompt>cd workdirectory
```

```
prompt>runme
```

UNIX

```
ksh prompt>cd workdirectory
```

```
ksh prompt>./runme.ksh
```

The Java Simpapp sample application runs and prints the following messages:

```
Testing simpapp
    cleaned up
    prepared
    built
    loaded ubb
    booted
    ran
    shutdown
    saved results
  PASSED
```

Note: After executing the `runme` command, you may get a message indicating the `Host`, `Port`, and `IPCKEY` parameters in the `UBBCONFIG` file conflict with an existing `UBBCONFIG` file. If this occurs, you need to set these parameters to different values to get the Java Simpapp sample application running on your machine. See the `Readme.txt` file for information about how to set these parameters.

The `runme` command starts the following application processes:

- `TMSYSEVT`

  The BEA TUXEDO system event broker.

- `TMFFNAME`

  The following three `TMFFNAME` server processes are started:

  - The `TMFFNAME` server process started with the `-N` and `-M` options is the master NameManager service. The NameManager service maintains a mapping of the application-supplied names to object references.

- ♦ The `TMFFNAME` server process started with only the `-N` option is the slave NameManager service.
- ♦ The `TMFFNAME` server process started with the `-F` option contains the FactoryFinder object.

♦ `JavaServer`

The Java Simpapp sample application server process. The JavaServer process has one option, `simple`, which is the Java Archive (JAR) file that was created for the application.

♦ `ISL`

The IIOP Listener process.

Table 2-4 lists the files in the work directory generated by the `runme` command.

Table 2-4 Files Generated by the `runme` Command

File	Description
`SimpleFactory.java`	Generated by the `m3idltojava` command for the `SimpleFactory` interface. The `SimpleFactory` interface contains the Java version of the OMG IDL interface. It extends `org.omg.CORBA.Object`.
`SimpleFactoryHolder.java`	Generated by the `m3idltojava` command for the `SimpleFactory` interface.This class holds a public instance member of type `SimpleFactory`. The class provides operations for `out` and `inout` arguments that are included in CORBA, but that do not map exactly to Java.
`SimpleFactoryHelper.java`	Generated by the `m3idltojava` command for the `SimpleFactory` interface. This class provides auxiliary functionality, notably the `narrow` method.
`_SimpleFactoryStub.java`	Generated by the `m3idltojava` command for the `SimpleFactory` interface. This class is the client stub that implements the `SimpleFactory.java` interface.

Table 2-4 Files Generated by the `runme` Command

File	Description
`_SimpleFactoryImplBase.java`	Generated by the `m3idltojava` command for the `SimpleFactory` interface. This abstract class is the server skeleton. It implements the `SimpleFactory.java` interface. The user-written server class `SimpleFactoryImpl` extends `_SimpleFactoryImplBase`.
`Simple.java`	Generated by the `m3idltojava` command for the `Simple` interface. The `Simple` interface contains the Java version of the OMG IDL interface. It extends `org.omg.CORBA.Object`.
`SimpleHolder.java`	Generated by the `m3idltojava` command for the `Simple` interface.This class holds a public instance member of type `Simple`. The class provides operations for out and inout arguments that CORBA has but that do not match exactly to Java.
`SimpleHelper.java`	Generated by the `m3idltojava` command for the `Simple` interface. This class provides auxiliary functionality, notably the `narrow` method.
`_SimpleStub.java`	Generated by the `m3idltojava` command for the `Simple` interface. This class is the client stub that implements the `Simple.java` interface.
`_SimpleImplBase.java`	Generated by the `m3idltojava` command for the `Simple` interface. This abstract class is the server skeleton. It implements the `Simple.java` interface. The user-written server class `SimpleImpl` extends `_SimpleImplBase`.
`Simple.ser`	The Server Descriptor File generated by the `buildjobjserver` command in the `runme` command.
`Simple.jar`	The server Java Archive file generated by the `buildjavaserver` command in the `runme` command.

Table 2-4 Files Generated by the `runme` Command

File	Description
`SimpleClient.jar`	The Java Archive file for the client application. It can be used to verify. This file is used during the installation of the WLE software to insure the client application is installed properly. For information about verifying the installation of the WLE software, see *Installing the WebLogic Enterprise Software*.
`.adm/.keybd`	A file that contains the security encryption key database. The subdirectory is created by the `tmloadcf` command in the `runme` command.
`results`	A directory generated by the `runme` command.

Table 2-5 lists files in the `results` directory generated by the `runme` command.

Table 2-5 Files in the `results` Directory Generated by the runme Command

File	Description
`input`	Contains the input that the `runme` command provides to the Java client application.
`output`	Contains the output produced when the `runme` command executes the Java client application.
`expected_output`	Contains the output that is expected when the Java client application is executed by the `runme` command. The data in the `output` file is compared to the data in the `expected_output` file to determine whether or not the test passed or failed.
`log`	Contains the output generated by the `runme` command. If the `runme` command fails, check this file for errors.

Table 2-5 Files in the `results` Directory Generated by the runme Command

File	Description
`setenv.cmd`	Contains the commands to set the environment variables needed to build and run the Java Simpapp sample application on the Windows NT operating system platform.
`setenv.ksh`	Contains the commands to set the environment variables needed to build and run the Java Simpapp sample application on the UNIX operating system platform.
`stderr`	Generated by the `tmboot` command, which is executed by the `runme` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.err.println` method sends the output to the `stderr` file instead of to the `ULOG` file.
`stdout`	Generated by the `tmboot` command, which is executed by the `runme` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.out.println` method sends the output to the `stdout` file instead of to the `ULOG` file.
`tmsysevt.dat`	Contains filtering and notification rules used by the TMSYSEVT (system event reporting) process. This file is generated by the `tmboot` command in the `runme` command.
`tuxconfig`	A binary version of the `UBBCONFIG` file.
`ubb`	The `UBBCONFIG` file for the Java Simpapp sample application.
`ULOG.<date>`	A log file that contains messages generated by the `tmboot` command.

Using the Java Simpapp Sample Application

This section describes how to use the Java Simpapp sample application after the `runme` command is executed.

Run the Java server application in the Java Simpapp sample application, as follows:

Windows NT

```
prompt>tmboot
```

UNIX

```
ksh prompt>tmboot
```

Run the Java client application in the Java Simpapp sample application, as follows:

Windows NT

```
prompt>java -classpath .;%TUXDIR%\udataobj\java\jdk\m3envobj.jar
-DTOBJADDR=%TOBJADDR% SimpleClient
String?
Hello World
HELLO WORLD
hello world
```

UNIX

```
ksh prompt>java -classpath .:$TUXDIR/udataobj/java/jdk\
/m3envobj.jar -DTOBJADDR=$TOBJADDR SimpleClient
String?
Hello World
HELLO WORLD
hello world
```

Note: The Java Simpapp sample client application uses the client-only JAR file `m3envobj.jar`. However, you could also use the `m3.jar` file to run the client application.

Using the C++ Client Application with the Java Simpapp Sample Application

A C++ client application is provided with the Java Simpapp sample application to demonstrate interoperabililty between a Java server application and a C++ client application. This section describes the process of building and running the C++ client application.

Build the C++ client application in the Java Simpapp sample application as follows:

1. Copy the files from the following directory to a work directory:

   Windows NT

   ```
   \WLEdir\samples\CORBA\simpapp_java
   ```

   UNIX

   ```
   /usr/local/WLEdir/samples/corba/simapp_java
   ```

Note: The work directory for the Java Simpapp sample application cannot be the same as the work directory for the C++ Simpapp sample application.

2. Change the protection on the files using the following commands:

   Windows NT

   ```
   prompt>attrib -r drive:\workdirectory\*.*
   ```

   UNIX

   ```
   prompt>/bin/ksh
   ksh prompt>chmod u+w /workdirectory/*.*
   ```

3. Make sure the UNIX `make` command or the Windows NT `nmake` command is in the path of your machine.

4. Set the `M3SIMPDIR` environment variable to your work directory.

5. Build the C++ client application, as follows:

   Windows NT

   ```
   prompt>cd %M3SIMPDIR
   ```

```
prompt>nmake -f makefile.nt simple_client.exe
```

UNIX

```
ksh prompt>cd %M3SIMPDIR
```

```
ksh prompt>make -f makefile.mk simple_client
```

Run the Java server application in the Java Simpapp sample application, as follows:

Windows NT

```
prompt>tmboot
```

UNIX

```
ksh prompt>tmboot
```

Run the C++ client application in the Java Simpapp sample application, as follows:

Windows NT

```
prompt>%M3SIMPDIR%\simple_client
String? Hello
HELLO
hello
```

UNIX

```
ksh prompt>$M3SIMPDIR/simple_client
String? Hello
HELLO
hello
```

Stopping the Java Simpapp Sample Application

Before using another sample application, enter the following commands to stop the Java Simpapp sample application and to remove unnecessary files from the work directory:

Windows NT

```
prompt>tmshutdown -y
```

```
prompt>nmake -f makefile.nt clean
```

UNIX

```
ksh prompt>tmshutdown -y
```

```
ksh prompt>make -f makefile.mk clean
```

CHAPTER

3 The JDBC Bankapp Sample Application

This chapter discusses the following topics:

- How the JDBC Bankapp sample application works
- The development process for the JDBC Bankapp sample application
- Setting up the database for the JDBC Bankapp sample application
- Building the JDBC Bankapp sample application
- Compiling the client and server applications
- Initializing the database
- Starting the server application in the JDBC Bankapp sample application
- Files generated by the JDBC Bankapp sample application
- Starting the Automatic Teller Machine (ATM) client application in the JDBC Bankapp sample application
- Stopping the JDBC Bankapp sample application
- Using the ATM client application

Refer to the `Readme.txt` file in the `\WLEdir\samples\corba\bankapp_java` directory for troubleshooting information and for the latest information about using the JDBC Bankapp sample application.

How the JDBC Bankapp Sample Application Works

The JDBC Bankapp sample application implements an automatic teller machine (ATM) interface and uses Java Database Connectivity (JDBC) to access a database that stores account and customer information. The JDBC Bankapp sample application consists of a Java server application that contains the following objects:

- `TellerFactory` object

  The `TellerFactory` object creates the object references to the `Teller` object.

- `Teller` object

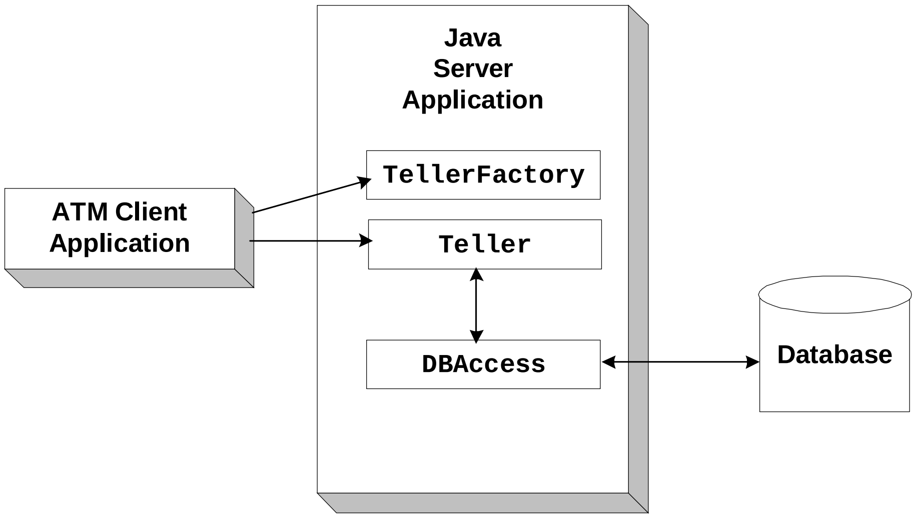
  The `Teller` object receives and processes requests for banking operations from the ATM client application.

- `DBAccesss` object

  The `DBAccess` object receives and processes requests from the `Teller` object to the database.

Figure 3-1 illustrates how the JDBC Bankapp sample application works.

Figure 3-1 The JDBC Bankapp Sample Application



The JDBC Bankapp sample application demonstrates how to implement JDBC database connection pooling running in a multithreaded server application. In the JDBC Bankapp sample application, a pool of database connections is created when the Java server application is initialized. All `DBAccess` objects share this pool.

A minimum number of database connections is established when the server application is initialized; the number of connections is increased on demand. When a worker thread receives a request for a `DBAccess` object, the corresponding `DBAccess` method gets an available database connection from the pool. When the call to the `DBAccess` method completes, the database connection is returned to the pool. If there is no database connection available and the maximum number of database connections has been established, the worker thread waits until a database connection becomes available.

The Development Process for the JDBC Bankapp Sample Application

This section describes the development process for the JDBC Bankapp sample application.

Note: The steps in this section have been done for you and are included in the JDBC Bankapp sample application.

Object Management Group (OMG) Interface Definition Language (IDL)

The OMG IDL for the JDBC Bankapp sample application defines the following CORBA interfaces:

Interface	Description	Methods
`TellerFactory`	Creates object references to the `Teller` object	`create_Teller()`

`Teller`	Performs banking operations	`verify_pin_number()`
		`deposit()`
		`withdraw()`
		`inquiry()`
		`transfer()`
		`report()`

`DBAccess`	Accesses the Oracle database on behalf of the `Teller` object	`get_valid_accounts()`
		`read_account()`
		`update_account()`
		`transfer_funds()`

Listing 3-1 shows the `BankApp.idl` file that defines the `TellerFactory` and `Teller` interfaces in the JDBC Bankapp sample application. A copy of this file is included in the directory for the JDBC Bankapp sample application.

Listing 3-1 OMG IDL Code for the `TellerFactory` and `Teller` Interfaces

```
#pragma prefix "beasys.com"
#pragma javaPackage "com.beasys.samples"

#include "Bank.idl"

module BankApp{
               exception IOException {};
               exception TellerInsufficentFunds();

               struct     BalanceAmounts{
                    float fromAccount;
                    float toAccount;
               };

               struct     TellerActivity {
                    long  totalRequests;
                    long  totalSuccesses;
                    long  totalFailures;
                    float currentBalance;
               };

               //Process Object
               interface Teller  {
                     void verify_pin_number(in short  pinNo,
                                      out Bank::CustAccounts accounts)
                               raises(Bank::PinNumberNotFound, IOException);
                     float deposit(in long accountNo, in float amount)
                               raises(Bank::AccountRecordNotFound,IOException);
                     float withdraw(in long accountNo, in float amount)
                               raises(Bank::AccountRecordNotFound,
                                       Bank::InsufficentFunds,
                                       IOException, TellerInsufficientFunds);
                     float inquiry(in long accountNo)
                               raises(Bank::AccountRecordNotFound, IOException);
                     void transfer(in long fromAccountNo,
                                   in long toAccountNo,in float amount,
                                   out BalanceAmounts balAmounts)
                               raises(Bank::AccountRecordNotFound,
                                      Bank::InsufficientFunds,
                                      IOException);
                     void report(out TellerActivity tellerData)
```

```
                        raises(IOException);
        };

        interface TellerFactory{
                Teller createTeller(in string tellerName);
        };

};
```

Listing 3-2 shows the `BankDB.idl` file that defines the `DBAccess` interface in the JDBC Bankapp sample application. A copy of this file is included in the directory for the JDBC Bankapp sample application.

Listing 3-2 OMG IDL Code for the `DBAccess` Interface

```
#pragma prefix "beasys.com"
#pragma javaPackage "com.beasys.samples"

#include "Bank.idl"

module BankDB{
             struct AccountData{
                  long accountID;
                  float balance;
             };

             interface DBAccess{
                  void get_valid_accounts(in short, pinNo,
                                        out Bank::CustAccounts accounts)
                          raises(Bank::DatabaseException,
                                 Bank::PinNumberNotFound);
                  void read_account(inout AccountData data)
                          raises(Bank::DatabaseException,
                                 Bank::AccountRecordNotFound);
                  void update_account(inout AccountData data)
                          raises(Bank::DatabaseException,
                                 Bank::AccountRecordNotFound,
                                 Bank::InsufficientFunds);
                  void transfer_funds(in float_amount,
                                        inout AccountData fromAcct,
                                        inout AccountData toAcct,
                          raises(Bank::DatabaseException,
                                     Bank::AccountRecordNotFound,
                                     Bank::InsufficientFunds);
             };

};
```

Listing 3-3 shows the `Bank.idl` file that defines common exceptions and structures. It is included by both `BankApp.idl` and `BankDB.idl`. A copy of this file is included in the directory for the JDBC Bankapp sample application.

Listing 3-3 OMG IDL Code for the Exceptions and Structures in JDBC Bankapp

```
#pragma prefix "beasys.com"
#pragma javaPackage "com.beasys.samples"

module Bank{

            exception DataBaseException {};
            exception PinNumberNotFound ();
            exception AccountRecordNotFound ();
            exception InsufficientFunds ();

            struct CustAccounts{
                  long checkingAccountID;
                  long savingsAccountID;
           };

};
```

The Client Application

During the development of the client application, you would write Java code that does the following:

- Initializes the ORB
- Uses the Bootstrap environmental object to establish communication with the WebLogic Enterprise (WLE) domain
- Resolves initial references to the FactoryFinder environmental object
- Uses a factory to get an object reference for the `Teller` object
- Invokes the `verify_pin_number`, `deposit`, `withdraw`, `inquiry`, `transfer`, and `report` methods on the `Teller` object

A Java client application, referred to as the ATM client application, is included in the JDBC Bankapp sample application. For more information about writing Java client applications, see *Creating Client Applications*.

The Server Application

During the development of the server application, you would write the following:

- The Server object, which initializes the server application in the JDBC Bankapp sample application and registers a factory for the `Teller` object with the WLE domain.
- The implementations for the methods of the `Teller` and `DBAccess` objects.

  The implementations for the `Teller` object include invoking operations on the `DBAccess` object.

  Because the `Teller` object has durable state (for example, ATM statistics) that is stored in an external source (a flat file), the method implementations must also include the `activate_object` and `deactivate_object` methods to ensure the `Teller` object is initialized with its state.

The JDBC Bankapp server application is configured to be multithreaded. Writing a multithreaded WLE Java server application is the same as writing a single-threaded Java server application; you cannot establish multiple threads programmatically in your object implementations. Instead, you establish the number of threads for a Java server application in the `UBBCONFIG` file. For information about writing Java server applications and using threads in Java server applications, see *Creating Java Server Applications*.

The Server Description File

During development, you create a Server Description File (`BankApp.xml`) that defines the activation and transaction policies for the `TellerFactory`, `Teller`, and `DBAccess` interfaces. For the JDBC Bankapp sample application, the objects have the following activation and transaction policies:

Interface	Activation Policy	Transaction Policy
`TellerFactory`	Process	Never
`Teller`	Method	Never
`DBAccess`	Method	Never

A Server Description File for the JDBC Bankapp sample application is provided. For information about creating Server Description Files and defining activation and transaction policies on objects, see *Creating Java Server Applications*.

The UBBCONFIG File

When using the WLE software, the server application is represented by a Java Archive (JAR). The JAR must be loaded into the Java Virtual Machine (JVM) to be executed. The JVM must execute in a WLE server application to be integrated in an WLE application. By default, the server application that loads the JVM is called `JavaServer`. You include the options to start `JavaServer` in the `Servers` section of the application's `UBBCONFIG` file.

If your Java server application is multithreaded, you can establish the number of threads by using the command-line option (`CLOPT`) `-M` in the `SERVERS` section of the `UBBCONFIG` file. In the following example, the `-M 100` option enables multithreading for the JavaServer and specifies 100 as the maximum number of worker threads that a particular instance of JavaServer can support. The largest number that you can specify is 500.

```
JavaServer
  SRVGRP = BANK_GROUP1
  SRVID = 2
  CLOPT = "-A -- -M 100 Bankapp.jar TellerFactory_1"
  RESTART = N
```

You also need to set the `MAXACCESSERS` parameter in the `RESOURCES` section of the `UBBCONFIG` file to account for the number of worker threads that each server application is configured to run. The `MAXACCESSERS` parameter specifies the number of processes that can attach to a WLE application.

For the JDBC Bankapp sample application, you need to include the following information on the command-line option (`CLOPT`) in the `SERVERS` section of the `UBBCONFIG` file:

- The type of JDBC driver: either `JdbcOracle2` for the jdbcKona/Oracle driver, or `JdbcMSSQL4` for the jdbcKona/MSSQLServer driver
- Either the connection string you defined for the Oracle database, or the name of the machine where the Microsoft SQL Server database is installed

- Either the user id for the Oracle database, or the user name you defined for the master instance of the Microsoft SQL Server database
- Either the password for the Oracle database, or the password you defined for the master instance of the Microsoft SQL Server database
- The initial number of database connections in the pool
- The maximum number of database connections in the pool

For example:

```
JavaServer
   SRVGRP  = BANK_GROUP1
   SRVID   = 2
   CLOPT   = "-A -- -M 10 BankApp.jar TellerFactory_1 jdbcOracle2  Beq-Local
   scott tiger 2 5"
   RESTART = N
```

Note: The information for the `CLOPT` parameter needs to be entered on one line.

For information about starting the `JavaServer` and defining parameters in the `UBBCONFIG` file, see the *Administration Guide*.

Setting Up the Database for the JDBC Bankapp Sample Application

The JDBC Bankapp sample application uses a database to store all the bank data. You can use either the Oracle or the Microsoft SQL Server database with the JDBC Bankapp sample application.

If you are using Oracle as the database for the JDBC Bankapp sample application, you need to install the following software:

- Visual C++ Version 5.0 with Service Pack for Visual Studio (Windows NT only)
- Sun SparcWorks Compiler 4.2 (Solaris only)
- Oracle Version 7.3.4

If you are using the Microsoft SQL Server as the database for the JDBC Bankapp sample application, you need to install the following software:

- Visual C++ Version 5.0 with Service Pack for Visual Studio (Windows NT only)
- Sun SparcWorks Compiler 4.2 (Solaris only)
- Microsoft SQL Server Version 6.50 (Service Pack 3)

Before you can build and run the JDBC Bankapp sample application, you need to follow the steps in the product documentation to install the desired database.

When using the Microsoft SQL Server database, you use the master database instance. You need the name of the machine where the Microsoft SQL Server database is installed and the user name and password you defined for the master instance of the Microsoft SQL Server database. Refer to the Microsoft product documentation for details about obtaining this information.

When using the Oracle database, you use the default database created by the Oracle installation program. You need the connection string you defined for the Oracle database and the default user id and password. Refer to the Oracle product documentation for details about obtaining this information.

The jdbcKona/Oracle and jdbcKona/MSSQLServer4 drivers are installed as part of the WLE installation. For more information about the jdbcKona drivers, refer to the *JDBC Driver Programming Reference* and *Installing the WebLogic Enterprise Software*.

Note: The jdbcKona/Oracle driver supports only Oracle Version 7.3.4 or higher.

Building the JDBC Bankapp Sample Application

To build the JDBC Bankapp sample application:

1. Copy the files for the JDBC Bankapp sample application into a work directory.
2. Change the protection attribute on the files for the JDBC Bankapp sample application.

3. Verify the settings of the environment variables.
4. Run the `setupJ` command.
5. Load the `UBBCONFIG` file.

The following sections describe these steps.

Copying the Files for the JDBC Bankapp Sample Application into a Work Directory

You need to copy the files for the JDBC Bankapp sample application into a work directory on your local machine. The files for the JDBC Bankapp sample application are located in the following directories:

Windows NT

`drive:``\WLEdir\samples\corba\bankapp_java\JDBC`

`drive:``\WLEdir\samples\corba\bankapp_java\client`

`drive:``\WLEdir\samples\corba\bankapp_java\shared`

UNIX

`/usr/local/WLEdir/samples/corba/bankapp_java/JDBC`

`/usr/local/WLEdir/samples/corba/bankapp_java/client`

`/usr/local/WLEdir/samples/corba/bankapp_java/shared`

The directories contain the following:

- `JDBC`—contains the source files and commands needed to build and run the JDBC Bankapp sample application.
- `client`—contains files for the ATM client application. The `images` subdirectory contains `.gif` files used by the graphical user interface in the ATM client application.
- `shared`—contains common files for the JDBC Bankapp and XA Bankapp sample applications.

You need to manually copy only the files in the `\JDBC` directory. The other sample application files are automatically copied from the `\client` and `\shared` directories when you execute the `setupJ` command. For example:

Windows NT

```
prompt> cd c:\mysamples\bankapp_java\JDBC
```

```
prompt> copy c:\WLEdir\samples\corba\bankapp_java\JDBC\*
```

UNIX

```
ksh prompt> cd /usr/mysamples/bankapp_java/JDBC/*
```

```
ksh prompt> cp $TUXDIR/samples/bankapp_java/JDBC/* .
```

Note: You cannot run the JDBC Bankapp sample application in the same work directory as the XA Bankapp sample application, because some of the files for the JDBC Bankapp sample application have the same name as files for the XA Bankapp sample application.

You will use the files listed in Table 3-1 to build and run the JDBC Bankapp sample application.

Table 3-1 Files Included in the JDBC Bankapp Sample Application

File	Description
`Bank.idl`	The OMG IDL code that declares common structures and extensions for the JDBC Bankapp sample application.
`BankApp.idl`	The OMG IDL code that declares the `TellerFactory` and `Teller` interfaces.
`BankDB.idl`	The OMG IDL code that declares the `DBAccess` interface.
`TellerFactoryImpl.java`	The Java source code that implements the `createTeller` method. This file is in the `com.beasys.samples` package. It is automatically moved to the `com/beasys/samples` directory by the `setupJ` command.

Table 3-1 Files Included in the JDBC Bankapp Sample Application

File	Description
`TellerImpl.java`	The Java source code that implements the `verify`, `deposit`, `withdraw`, `inquiry`, `transfer`, and `report` methods. This file is in the `com.beasys.samples` package. It is automatically moved to the `com/beasys/samples` directory by the `setupJ` command.
`BankAppServerImpl.java`	The Java source code that overrides the `Server.initialize` and `Server.release` methods.
`DBAccessImpl.java`	The Java source code that implements the `get_valid_accounts`, `read_account`, `update_account`, and `transfer` methods. This file is in the `com.beasys.samples` package. It is automatically moved to the `com/beasys/samples` directory by the `setupJ` command.
`Atm.java`	The Java source code for the ATM client application.
`BankStats.java`	Contains methods to initialize, read from, and write to the flat file that contains the ATM statistics.
`BankApp.xml`	The Server Description File used to associate activation and transaction policy values with CORBA interfaces.
`InitDB.java`	A Java program that initializes the database and ensures that JDBC is working properly.
`ConnectionPool.java`	Implements the database connection pooling mechanism.
`setupJ.cmd`	The Windows NT batch file that builds and runs the JDBC Bankapp sample application.
`setupJ.ksh`	The UNIX Korn shell script that builds and runs the JDBC Bankapp sample application.

Table 3-1 Files Included in the JDBC Bankapp Sample Application

File	Description
`makefileJ.mk`	The make file for the JDBC Bankapp sample application on the UNIX operating system. The UNIX `make` command needs to be in the path of your machine.
`makefileJ.nt`	The make file for the JDBC Bankapp sample application on the Windows NT operating system. The Windows NT `nmake` command needs to be in the path of your machine.
`Readme.txt`	The file that provides the latest information about building and running the JDBC Bankapp sample application.

Changing the Protection Attribute on the Files for the JDBC Bankapp Sample Application

During the installation of the WLE software, the files for the JDBC Bankapp sample application are marked read-only. Before you can edit or build the files in the JDBC Bankapp sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

Windows NT

```
prompt>attrib -r drive:\workdirectory\*.*
```

UNIX

```
prompt>/bin/ksh
```

```
ksh prompt>chmod u+w /workdirectory/*.*
```

Verifying the Settings of the Environment Variables

Before building and running the JDBC Bankapp sample application, you need to ensure that certain environment variables are set on your system. In most cases, these environment variables are set as part of the installation procedure. However, you need to check the environment variables to ensure they reflect correct information.

Table 3-2 lists the environment variables required to run the JDBC Bankapp sample application.

Table 3-2 Required Environment Variables for the JDBC Bankapp Sample Application

Environment Variable	Description

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

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

`ORACLE_HOME`	The directory path where you installed the Oracle software. For example: **UNIX** `ORACLE_HOME=/usr/local/oracle` **Note:** This environment variable applies only if you are using the Oracle database on the Solaris operating system.

To verify that the information defined during installation is correct:

Windows NT

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`.

UNIX

```
ksh prompt>printenv TUXDIR

ksh prompt>printenv JAVA_HOME

ksh prompt>printenv ORACLE_HOME
```

To change the settings:

Windows NT

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. In the Value field, enter the correct information for the environment variable.
3. Click OK to save the changes.

UNIX

```
ksh prompt>TUXDIR=directorypath; export TUXDIR

ksh prompt>JAVA_HOME=directorypath; export JAVA_HOME

ksh prompt>JAVA_HOME=directorypath; export ORACLE_HOME
```

Note: If you are running multiple WLE applications concurrently on the same machine, you also need to set the `IPCKEY` and `PORT` environment variables. See the `Readme.txt` file for information about how to set these environment variables.

Running the setupJ Command

The `setupJ` command automates the following steps:

1. Copying required files from the `\client` and `\shared` directories
2. Setting the `PATH`, `TOBJADDR`, `APPDIR`, `TUXCONFIG`, and `CLASSPATH` system environment variables
3. Creating the `UBBCONFIG` file
4. Creating a `setenvJ.cmd` or `setenvJ.ksh` file that can be used to reset the system environment variables

Enter the `setupJ` command, as follows:

Windows NT

```
prompt>cd c:\mysamples\bankapp_java\JDBC

prompt>setupJ
```

UNIX

```
prompt>/bin/ksh

prompt>cd /usr/mysamples/bankapp_java/JDBC

prompt>. ./setupJ.ksh
```

Loading the UBBCONFIG File

Use the following command to load the `UBBCONFIG` file:

```
prompt>tmloadcf -y ubb_jdbc
```

Compiling the Client and Server Applications

The directory for the JDBC Bankapp sample application contains a make file that builds the client and server sample applications. During development, you use the `buildjavaserver` command to build the server application, and your Java product's development commands to build the client application. However, for the JDBC Bankapp sample application, these steps are included in the make file.

Use the following commands to build the client and server applications in the JDBC Bankapp sample application:

Windows NT

```
prompt>nmake -f makefileJ.nt
```

UNIX

```
prompt>make -f makefileJ.mk
```

Note: If you are using the Microsoft SQL Server database, you need to manually modify the JavaServer line in the UBBCONFIG file. For information about the JavaServer information in the UBBCONFIG file, see the section "Starting the Server Application in the JDBC Bankapp Sample Application."

Initializing the Database

To initialize the Oracle database using the default arguments, enter the following command:

```
prompt>java InitDB
```

To initialize the Oracle database with user-defined attributes, enter the following command:

```
prompt>java InitDB JdbcOracle2 connect_string username password
```

where

connect_string	Is the defined connection string for the instance of the Oracle database being used with the JDBC Bankapp sample application. The default value is `Beq-local`.
username	Is the user name you defined for the Oracle database. The default value is `scott`.
password	Is the password you defined for the Oracle database. The default value is `tiger`.

To initialize the Microsoft SQL Server database, enter the following command:

```
prompt>java InitDB JdbcMSSQL4 db_server username password
```

where

db_server	Is the name of the machine where the Microsoft SQL Server database is installed
username	Is the user name you defined for the `master` instance of the Microsoft SQL Server database
password	Is the password you defined for the `master` instance of the Microsoft SQL Server database

Starting the Server Application in the JDBC Bankapp Sample Application

Start the server application in the JDBC Bankapp sample application by entering the following command:

```
prompt>tmboot -y
```

The `tmboot` command starts the following application processes:

- `TMSYSEVT`

  The BEA TUXEDO system event broker.

- `TMFFNAME`

  Three `TMFFNAME` server processes are started:

  - The `TMFFNAME` server process started with the `-N` and `-M` options is the master NameManager service. The NameManager service maintains a mapping of the application-supplied names to object references.
  - The `TMFFNAME` server process started with the `-N` option is the slave NameManager service.
  - The `TMFFNAME` server process started with the `-F` option contains the FactoryFinder object.

- `JavaServer`

  The server process that implements the `TellerFactory`, `Teller,` and `DBAccess` interfaces.

- `ISL`

  The IIOP Listener process.

Files Generated by the JDBC Bankapp Sample Application

Table 3-3 lists the files generated by the JDBC Bankapp sample application.

Table 3-3 Files Generated by the JDBC Bankapp Sample Application

File	Description
`ubb_jdbc`	The `UBBCONFIG` file for the JDBC Bankapp sample application. This file is generated by the `setupJ` command.

Table 3-3 Files Generated by the JDBC Bankapp Sample Application

File	Description
`setenvJ.cmd` and `setenvJ.ksh`	Contains the commands to set the environment variables needed to build and run the JDBC Bankapp sample application. `setenvJ.cmd` is the Windows NT version and `setenvJ.ksh` is the UNIX Korn shell version of the file.
`tuxconfig`	A binary version of the `UBBCONFIG` file. Generated by the `tmloadcf` command.
`ULOG.<`*`date`*`>`	A log file that contains messages generated by the `tmboot` command. The log file also contains messages generated by the server applications and by the `tmshutdown` command.
`.adm/.keybd`	A file that contains the security encryption key database. The subdirectory is created by the `tmloadcf` command.

`Atm$1.class`
`Atm.class`
`AtmAppletStub.class`
`AtmArrow.class`
`AtmButton.class`
`AtmCenterTextCanvas.class`
`AtmClock.class`
`AtmScreen.class`
`AtmServices.class`
`AtmStatus.class`
	Used by the Java client application. Created when the `Atm.java` file is compiled.

`ConnectionPool.class`	Defines how database pooling works in the JDBC sample application.
`InitDB.class`	Initializes the database used by the JDBC Bankapp sample application. Created when `InitDB.java` is compiled.

Table 3-3 Files Generated by the JDBC Bankapp Sample Application

File	Description

`AccountRecordNotFound.java`
`AccountRecordNotFoundHelper.java`
`AccountRecordNotFoundHolder.java`
`CustAccounts.java`
`CustAccountsHelper.java`
`CustAccountsHolder.java`
`DataBaseException.java`
`DataBaseExceptionHelper.java`
`DataBaseExceptionHolder.java`
`InsufficientFunds.java`
`InsufficientFundsHelper.java`
`InsufficientFundsHolder.java`
`PinNumberNotFound.java`
`PinNumberNotFoundHelper.java`
`PinNumberNotFoundHolder.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `Bank.idl` file. These files are created in the `com/beasys/samples/Bank` directory.

`BalanceAmounts.java`
`BalanceAmountsHelper.java`
`BalanceAmountsHolder.java`
`IOException.java`
`IOExceptionHelper.java`
`IOExceptionHolder.java`
`Teller.java`
`TellerActivity.java`
`TellerActivityHelper.java`
`TellerActivityHolder.java`
`TellerFactory.java`
`TellerFactoryHelper.java`
`TellerFactoryHolder.java`
`TellerInsufficientFunds.java`
`TellerInsufficientFundsHelper.java`
`TellerInsufficientFundsHolder.java`
`_TellerFactoryImplBase.java`
`_TellerFactoryStub.java`
`_TellerImplBase.java`
`_TellerStub.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `BankApp.idl` file. These files are created in the `com/beasys/samples/BankApp` subdirectory.

Table 3-3 Files Generated by the JDBC Bankapp Sample Application

File	Description

`AccountData.java`
`AccountDataHelper.java`
`AccountDataHolder.java`
`DBAccessHelper.java`
`DBAccessHolder.java`
`_DBAccessImplBase.java`
`_DBAccessStub.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `BankDB.idl` file. These files are created in the `com/beasys/samples/BankDB` subdirectory.

`Bankapp.ser` `Bankapp.jar`	The Server Descriptor File and Server Java Archive file generated by the `buildjavaserver` command in the make file.
`stderr`	Generated by the `tmboot` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.err.println` method sends the output to the `stderr` file instead of to the `ULOG` file.
`stdout`	Generated by the `tmboot` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.out.println` method sends the output to the `stdout` file instead of to the `ULOG` file.
`tmsysevt.dat`	Contains filtering and notification rules used by the TMSYSEVT (system event reporting) process. This file is generated by the `tmboot` command.

Starting the ATM Client Application in the JDBC Bankapp Sample Application

Start the ATM client application by entering the following command:

Note: The following command sets the Java CLASSPATH to include the current directory and the client JAR file (`m3envobj.jar`). The full WLE JAR file (`m3.jar`) can be specified instead of the client JAR file.

Windows NT

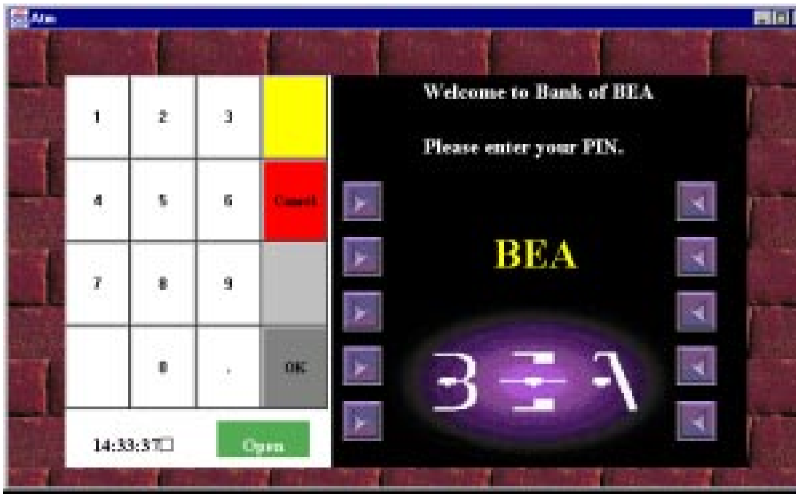
```
prompt>java -classpath .;%TUXDIR%\udataobj\java\jdk\m3envobj.jar
-DTOBJADDR=%TOBJADDR% Atm Teller1
```

UNIX

```
ksh prompt>java -classpath .:$TUXDIR/udataobj/java/jdk
/m3envobj.jar -DTOBJADDR=$TOBJADDR Atm Teller1
```

The GUI for the ATM client application appears. Figure 3-2 shows the GUI for the ATM client application.

Figure 3-2 GUI for ATM Client Application



Stopping the JDBC Bankapp Sample Application

Before using another sample application, enter the following commands to stop the JDBC Bankapp sample application and to remove unnecessary files from the work directory:

Windows NT

```
prompt>tmshutdown -y
```

```
prompt>nmake -f makefileJ.nt clean
```

UNIX

```
ksh prompt>tmshutdown -y
```

```
ksh prompt>make -f makefileJ.mk clean
```

Using the ATM Client Application

In the ATM client application, a customer enters a personal identification number (PIN) and performs one of the following banking operations:

- Withdraws money from the account
- Deposits money in the account
- Inquires about the balance of the account
- Transfers money between checking and savings accounts

One special PIN number (999) allows customers to receive statistics about the ATM machine. The following statistics are available:

- Total number of requests received by the ATM machine

  For example, an inquiry is one request, and a withdrawal is one request.
- Total number of successful requests
- Total number of failed requests

  For example, when a customer attempts to withdraw more money than is in his account, the request fails.
- Total amount of cash remaining in the ATM machine

  The ATM machine starts with $10,000 and the amount decreases with each withdrawal request.

Use the keypad in the ATM client application to enter a PIN and amounts for deposit, transfer, and withdrawal. Table 3-4 describes the functions available on the keypad in the ATM client application.

Table 3-4 Keypad Functions in the ATM Client Application

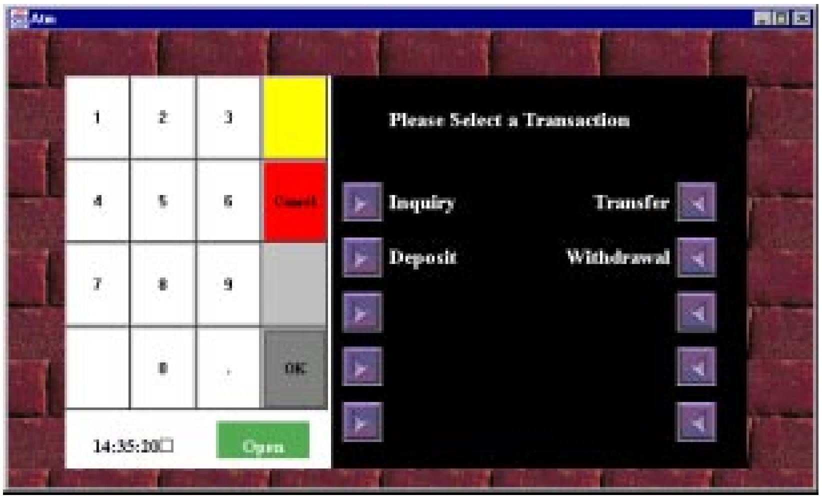
Key	Function
Cancel	Use this key to cancel the current operation and exit the view.
OK	Use this key to accept the entered data. After you enter a PIN or an amount for deposit, transfer, or withdrawal, you need to click the OK button to have the action take effect.
Numerics (0 through 9)	Use these keys to enter your PIN and an amount for deposit, transfer, and withdrawal amounts.
Period (.)	Use this key to enter decimal amounts for deposit, transfer, and withdrawal.

To use the ATM client application in the JDBC Bankapp sample application:

1. Enter one of the following PINs: 100, 110, 120, or 130.
2. Click OK.

   The Operations view appears. Figure 3-3 shows the Operations view in the ATM client application.

Figure 3-3 Operations View in the ATM Client Application



From the Operations view, you can perform the follow banking operations:

- ♦ Inquiry
- ♦ Transfer
- ♦ Deposit
- ♦ Withdrawl

3. Click the desired banking operation.
4. Click either the Checking Acct or Savings Acct button.
5. Enter a dollar amount.
6. Click OK.

   An updated account balance appears.

   Note: After you click OK, you cannot cancel the operation. If you enter an amount and then select Cancel, the ATM client application cancels your operation and displays the previous screen.

7. Click OK.
8. Click Cancel to return to the main window of the ATM client application.

CHAPTER

4 The XA Bankapp Sample Application

This chapter discusses the following topics:

- How the XA Bankapp sample application works
- Software prerequisites
- The development process for the XA Bankapp sample application
- Setting up the Oracle database for the XA Bankapp sample application
- Building the XA Bankapp sample application
- Compiling the client and server applications
- Initializing the Oracle database
- Starting the server applications in the XA Bankapp sample application
- Files generated by the XA Bankapp sample application
- Starting the Automatic Teller Machine (ATM) client application in the XA Bankapp sample application
- Stopping the XA Bankapp sample application
- Using the ATM client application

Refer to the `Readme.txt` file in the `\WLEdir\samples\corba\bankapp_java` directory for troubleshooting information and for the most recent information about using the XA Bankapp sample application.

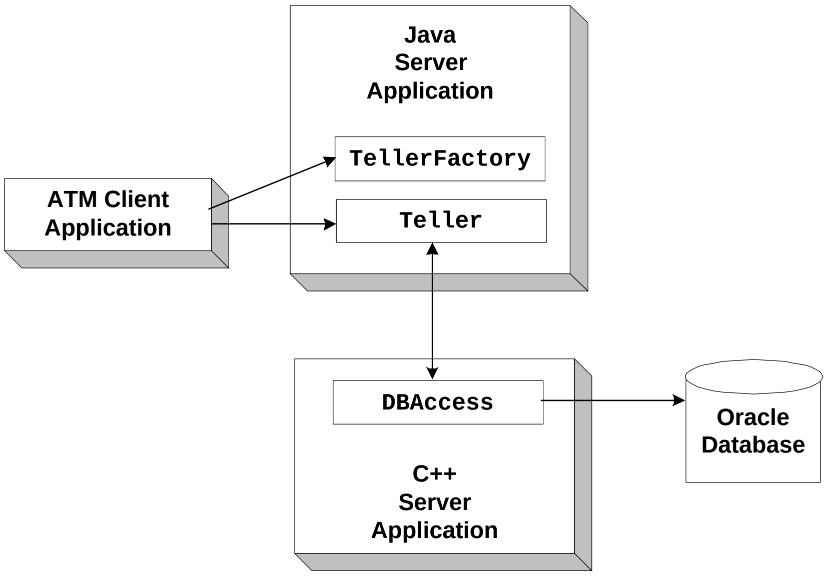
How the XA Bankapp Sample Application Works

The XA Bankapp sample application implements the same automatic teller machine (ATM) interface as the JDBC Bankapp sample application. However, the XA Bankapp sample application uses the Oracle XA library and the Weblogic Enterprise (WLE) Transaction Manager to coordinate transactions between the WLE application and the Oracle database that stores account and customer information. The XA Bankapp sample application consists of two server applications:

- A Java server application, which implements the `TellerFactory` and `Teller` objects
- A C++ server application, which processes requests on objects that implement the `DBAccesss` interface

Figure 4-1 illustrates how the XA Bankapp sample application works.

Figure 4-1 The XA Bankapp Sample Application



In the XA Bankapp sample application, transactions are started and stopped in the `Teller` object using the Java Transaction Service (JTS) protocol. In JDBC Bankapp, transactions are started and stopped in the DBAccess object using the Java Database Connectivity (JDBC) protocol.

In the XA Bankapp sample application, the `DBAccess` object is implemented in C++ instead of Java and resides in its own server application. The object reference for the `DBAccess` object is generated in its `Server::initialize` method and is registered with the FactoryFinder environmental object.

Software Prerequisites

To run the XA Bankapp sample application, you need to install the following software:

- Visual C++ Version 5.0 with Service Pack 3 for Visual Studio
- Oracle Version 7.3.4

The Development Process for the XA Bankapp Sample Application

This section describes the development process for the XA Bankapp sample application.

Note: The steps in this section have been done for you and are included in the XA Bankapp sample application.

Object Management Group (OMG) Interface Definition Language (IDL)

The `BankApp.idl` file used in the XA Bankapp sample application defines the `TellerFactory` and `Teller` interfaces and the `Bank.idl` file defines exceptions and structures. The `transfer_funds` interface has been removed from the `BankDB.idl` because transactions are now started and stopped by the `Teller` object.

The Client Application

The XA Bankapp sample application uses the same client application as the JDBC Bankapp sample application.

The Server Application

For the XA Bankapp sample application, you would write the following:

- The Java Server object, which initializes the Java server application in the XA Bankapp sample application and registers a factory for the Teller object with the WLE domain.
- The implementation for the methods of the Teller object. The implementation for the Teller object includes invoking operations on the DBAccess object. Because the Teller object has durable state (for example, ATM statistics), which is stored in an external source (a flat file), the method implementations must also include the `activate_object()` and `deactivate_object()` methods to ensure the Teller object is initialized with its state.
- The C++ server object which initializes the C++ server and registers the DBAccess object with the WLE domain.
- The implementation for the methods of the DBAccess object.

For information about writing server applications, see *Creating Java Server Applications* and *Creating C++ Server Applications*.

The Server Description File

During development, you create a Server Description File (`BankApp.xml`) that defines the activation and transaction policies for the `TellerFactory` and `Teller` objects. For the XA Bankapp sample application, the objects have the following activation and transaction policies:

Interface	**Activation Policy**	**Transaction Policy**
`TellerFactory`	Process	Never
`Teller`	Method	Never

A Server Description File for the XA Bankapp sample application is provided. For information about creating Server Description Files and defining activation and transaction policies on objects, see *Creating Java Server Applications*.

The Implementation Configuration File

When writing WLE C++ server applications, you create an Implementation Configuration File (ICF), which is similar to the Server Description File. This file has been created for you and defines an activation policy of `transaction` and a transaction policy of `always` for the `DBAccess` interface.

For information about creating ICF files and defining activation and transaction policies on objects, see *Creating C++ Server Applications*.

The UBBCONFIG File

During development, you need to include the following information in the `UBBCONFIG` file:

- ♦ The `OPENINFO` parameter, defined according to the `XA` parameter for the Oracle database. The `XA` parameter for the Oracle database is described in the "Developing and Installing Applications that Use the XA Libraries" section of the *Oracle7 Server Distributed Systems* manual.

- The pathname to the transaction log (`TLOG`) in the `TLOGDEVICE` parameter.

For information about the transaction log and defining parameters in the `UBBCONFIG` file, see the *Administration Guide*.

Setting Up the Database for the XA Bankapp Sample Application

The XA Bankapp sample application uses an Oracle database to store all the bank data. Before using the XA Bankapp sample application, you need to install the following Oracle components:

- Programmer/2000 Pro*C/C++, Version 2.2.4.0.0
- Oracle 7 Server, Version 7.3.4

Note: When installing the specified Oracle components, other Oracle components are also installed. However, you will not use these additional components with the XA Bankapp sample application.

You also need to start the Oracle database daemon and enable an XA resource manager.

For information about installing the Oracle database and performing the necessary setup tasks, see the product documentation for the Oracle database.

Building the XA Bankapp Sample Application

To build the XA Bankapp sample application:

1. Copy the files for the XA Bankapp sample application into a work directory.

2. Change the protection attribute on the files for the XA Bankapp sample application.
3. Verify the settings of the environment variables.
4. Run the `setupX` command.
5. Load the `UBBCONFIG` file.
6. Create a transaction log.

The following sections describe these steps.

Copying the Files for the XA Bankapp Sample Application into a Work Directory

You need to copy the files for the XA Bankapp sample application into a work directory on your local machine. The files for the XA Bankapp sample application are located in the following directories:

Windows NT

`drive:``\WLEdir\samples\corba\bankapp_java\XA`

`drive:``\WLEdir\samples\corba\bankapp_java\client`

`drive:``\WLEdir\samples\corba\bankapp_java\shared`

UNIX

`/usr/local/WLEdir/samples/corba/bankapp_java/XA`

`/usr/local/WLEdir/samples/corba/bankapp_java/client`

`/usr/local/WLEdir/samples/corba/bankapp_java/shared`

The directories contain the following:

- `XA`—contains the source files and commands needed to build and run the XA Bankapp sample application.
- `client`—contains files for the ATM client application. The `images` subdirectory contains `.gif` files used by the graphical user interface in the ATM client application.

- `shared`—contains common files for the JDBC Bankapp and XA Bankapp sample applications.

You only need to manually copy the files in the XA directory. The other files are automatically copied from the `\client` and `\shared` directories when you execute the `setupX` command. For example:

Windows NT

```
prompt> cd c:\mysamples\bankapp_xa\XA
```

```
prompt> copy c:\WLEdir\samples\corba\bankapp_xa\XA\*
```

UNIX

```
ksh prompt> cd /usr/mysamples/bankapp_xa/XA/*
```

```
ksh prompt> cp $TUXDIR/samples/bankapp_xa/XA/*
```

Note: You cannot run the XA Bankapp sample application in the same work directory as the JDBC Bankapp sample application, because some of the files for the JDBC Bankapp sample application have the same name as files for the XA Bankapp sample application.

You will use the files listed in Table 4-1 to build and run the XA Bankapp sample application.

Table 4-1 Files Included in the XA Bankapp Sample Application

File	Description
`Bank.idl`	The OMG IDL code that declares common structures and extensions for the XA Bankapp sample application.
`BankApp.idl`	The OMG IDL code that declares the `TellerFactory` and `Teller` interfaces.
`BankDB.idl`	The OMG IDL code that declares the `DBAccess` interface.
`BankDB.icf`	The ICF file that defines activation and transaction policies for the `DBAccess` interface.

Table 4-1 Files Included in the XA Bankapp Sample Application

File	Description
`BankDBServer.cpp`	The C++ source code that implements the `Server::initialize()` and `Server::release()` methods for the C++ server application.
`TellerFactoryImpl.java`	The Java source code that implements the `createTeller` method.
`TellerImpl.java`	The Java source code that implements the `verify`, `deposit`, `withdraw`, `inquiry`, `transfer`, and `report` methods. In addition, it includes a reference to the TransactionCurrent environmental object and invokes operations on the `DBAccess` object within a transaction.
`BankAppServerImpl.java`	The Java source code that overrides the `Server.initialize` and `Server.release` methods.
`Atm.java`	The Java source code for the ATM client application.
`BankStats.java`	Contains methods to initialize, read from, and write to the flat file that contains the ATM statistics.
`BankApp.xml`	The Server Description File used to associate activation and transaction policy values with CORBA interfaces.
`DBAccess_i.h` `DBAccess_i.pc`	The Oracle Pro*C/C++ code that implements the `DBAccess` interface.
`InitDB.sql`	The Oracle SQL *Plus script that creates and populates the database tables.
`setupX.cmd`	The Windows NT batch file that builds and runs the XA Bankapp sample application.
`setupX.ksh`	The UNIX Korn shell script that builds and runs the XA Bankapp sample application.

Table 4-1 Files Included in the XA Bankapp Sample Application

File	Description
`makefileX.mk`	The make file for the XA Bankapp sample application on the UNIX operating system. The UNIX `make` command needs to be in the path of your machine.
`makefileX.nt`	The make file for the XA Bankapp sample application on the Windows NT operating system. The Windows NT `nmake` command needs to be in the path of your machine.
`Readme.txt`	Provides the latest information about building and running the XA Bankapp sample application.

Changing the Protection Attribute on the Files for the XA Bankapp Sample Application

During the installation of theWLE software, the files for the XA Bankapp sample application are marked read-only. Before you can edit or build the files in the XA Bankapp sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

Windows NT

```
prompt>attrib -r drive:\workdirectory\*.*
```

UNIX

```
prompt>/bin/ksh

ksh prompt>chmod u+w /workdirectory/*.*
```

Verifying the Settings of the Environment Variables

Before building and running the XA Bankapp sample application, you need to ensure that certain environment variables are set on your system. In most cases, these environment variables are set as part of the installation procedure. However, you need to check the environment variables to ensure they reflect correct information.

Table 4-2 lists the environment variables required to run the XA Bankapp sample application.

Table 4-2 Required Environment Variables for the XA Bankapp Sample Application

Environment Variable	Description

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

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

`ORACLE_HOME`	The directory path where you installed the Oracle software. For example: `ORACLE_HOME=/usr/local/oracle` You need to set this environment variable on the Solaris operating system only.

To verify that the information defined during installation is correct:

Windows NT

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`, `ORACLE_HOME`, and `JAVA_HOME`.

UNIX

```
ksh prompt>printenv TUXDIR

ksh prompt>printenv JAVA_HOME

ksh prompt>printenv ORACLE_HOME
```

To change the settings:

Windows NT

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.

UNIX

```
ksh prompt>TUXDIR=directorypath; export TUXDIR

ksh prompt>JAVA_HOME=directorypath; export JAVA_HOME

ksh prompt>JAVA_HOME=directorypath; export ORACLE_HOME
```

Note: If you are running multiple WLE applications concurrently on the same machine, you also need to set the `IPCKEY` and `PORT` environment variables. See the `Readme.txt` file for information about how to set these environment variables.

Running the setupX Command

The `setupX` command automates the following steps:

1. Copying required files from the `\client` and `\shared` directories
2. Setting the `PATH`, `TOBJADDR`, `APPDIR`, `TUXCONFIG`, and `CLASSPATH` system environment variables
3. Creating the `UBBCONFIG` file
4. Creating a `setenvX.cmd` or `setenvX.ksh` file that can be used to reset the system environment variables

Enter the `setupX` command, as follows:

Windows NT

```
prompt> cd c:\mysamples\bankapp_xa\XA

prompt>setupX
```

UNIX

```
prompt>/bin/ksh

prompt> cd /usr/mysamples/bankapp_xa/XA/*

prompt>. ./setupX.ksh
```

Loading the UBBCONFIG File

Use the following command to load the `UBBCONFIG` file:

```
prompt>tmloadcf -y ubb_xa
```

Creating a Transaction Log

The transaction log records the transaction activities in a WLE session. During the development process, you need to define the location of the transaction log (specified by the `TLOGDEVICE` parameter) in the `UBBCONFIG` file. For the XA Bankapp sample application, the transaction log is placed in your work directory.

To open the transaction log for the XA Bankapp sample application:

1. Enter the following command to start the Interactive Administrative Interface:

   ```
   tmadmin
   ```

2. Enter the following command to create a transaction log:

   ```
   crdl -b blocks -z directorypath TLOG
   crlog -m SITE1
   ```

   where

   `blocks` specifies the number of blocks to be allocated for the transaction log and `directorypath` indicates the location of the transaction log. The `directorypath` option needs to match the location specified in the `TLOGDEVICE` parameter in the `UBBCONFIG` file. The following is an example of the command on Windows NT:

   ```
   crdl -b 500 -z c:\mysamples\bankapp_java\XA\TLOG
   ```

3. Enter `quit` to exit the Interactive Administrative Interface.

Compiling the Client and Server Applications

The directory for the XA Bankapp sample application contains a make file that builds the client and server applications. During the development process, you use the `buildjavaserver` command to build the server application, and your Java product's development commands to build the client application. However, for the XA Bankapp sample application, this step is included in the make file.

Use the following commands to build the client and server applications in the XA Bankapp sample application:

Windows NT

```
prompt>nmake -f makefileX.nt
```

UNIX

```
prompt>make -f makefileX.mk
```

Initializing the Oracle database

Use the following command to initialize the Oracle database used with the XA Bankapp sample application:

Windows NT

```
prompt>nmake -f makefileX.nt InitDB
```

UNIX

```
ksh prompt>make -f makefileX.mk InitDB
```

Starting the Server Application in the XA Bankapp Sample Application

Start the server application in the XA Bankapp sample application by entering the following command:

```
prompt>tmboot -y
```

The `tmboot` command starts the following application processes:

- `TMSYSEVT`

  The BEA TUXEDO system event broker.

- `TMFFNAME`

  Three `TMFFNAME` server processes are started:

  - The `TMFFNAME` server process with the `-N` and `-M` options is the master NameManager service. The NameManager service maintains a mapping of the application-supplied names to object references.
  - The `TMFFNAME` server process started with the `-N` option only is the slave NameManager service.
  - The `TMFFNAME` server process started with the `-F` option contains the FactoryFinder object.

- `TMS_ORA`

  The transaction manager service.

- `BankDataBase`

  The WLE server process that implements the `DBAccess` interface.

- `JavaServerXA`

  The server process that implements the `TellerFactory` and `Teller` interfaces. The JavaServer process has two options:

  - `BankApp.jar`, which is the Java Archive (JAR) file that was created by the `buildjavaserver` command.
  - `TellerFactory_1`, which is passed to the `Server.initialize()` method.

  `JavaServerXA` is a special version of `JavaServer` that uses the same XA switch as the `BankDataBase` server process. It is created by the `buildXAJS` command.

- `ISL`

  The IIOP Listener process.

Files Generated by the XA Bankapp Sample Application

Table 4-3 lists the files generated by the XA Bankapp sample application.

Table 4-3 Files Generated by the XA Bankapp Sample Application

File	Description
`ubb_xa`	The `UBBCONFIG` file for the XA Bankapp sample application. This file is generated by the `setupX` command.
`setenvX.cmd` and `setenvX.ksh`	Contains the commands to set the environment variables needed to build and run the XA Bankapp sample application. `setenvX.cmd` is the Windows NT version and `setenvX.ksh` is the UNIX Korn shell version of the file.
`tuxconfig`	A binary version of the `UBBCONFIG` file. Generated by the `tmloadcf` command.
`TLOG`	The transaction log.
`ULOG.<`*`date`*`>`	A log file that contains messages generated by the `tmboot` command. The log file also contains messages generated by the server applications and the `tmshutdown` command.
`.adm/.keybd`	A file that contains the security encryption key database. The subdirectory is created by the `tmloadcf` command.

Table 4-3 Files Generated by the XA Bankapp Sample Application

File	Description

`Atm$1.class`
`Atm.class`
`AtmAppletStub.class`
`AtmArrow.class`
`AtmButton.class`
`AtmCenterTextCanvas.class`
`AtmClock.class`
`AtmScreen.class`
`AtmServices.class`
`AtmStatus.class`
	Used by the Java client application. Created when the `Atm.java` file is compiled.

`AccountRecordNotFound.java`
`AccountRecordNotFoundHelper.java`
`AccountRecordNotFoundHolder.java`
`CustAccounts.java`
`CustAccountsHelper.java`
`CustAccountsHolder.java`
`DataBaseException.java`
`DataBaseExceptionHelper.java`
`DataBaseExceptionHolder.java`
`InsufficientFunds.java`
`InsufficientFundsHelper.java`
`InsufficientFundsHolder.java`
`PinNumberNotFound.java`
`PinNumberNotFoundHelper.java`
`PinNumberNotFoundHolder.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `Bank.idl` file. These files are created in the `\com\beasys\samples\Bank` subdirectory.

Table 4-3 Files Generated by the XA Bankapp Sample Application

File	Description

`BalanceAmounts.java`
`BalanceAmountsHelper.java`
`BalanceAmountsHolder.java`
`IOException.java`
`IOExceptionHelper.java`
`IOExceptionHolder.java`
`Teller.java`
`TellerActivity.java`
`TellerActivityHelper.java`
`TellerActivityHolder.java`
`TellerFactory.java`
`TellerFactoryHelper.java`
`TellerFactoryHolder.java`
`TellerInsufficientFunds.java`
`TellerInsufficientFundsHelper.java`
`TellerInsufficientFundsHolder.java`
`_TellerFactoryImplBase.java`
`_TellerFactoryStub.java`
`_TellerImplBase.java`
`_TellerStub.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `BankApp.idl` file. These files are created in the `\com\beasys\samples\BankApp` subdirectory.

`AccountData.java`
`AccountDataHelper.java`
`AccountDataHolder.java`
`DBAccessHelper.java`
`DBAccessHolder.java`
`_DBAccessImplBase.java`
`_DBAccessStub.java`
	Generated by the `m3idltojava` command for the interfaces defined in the `BankDB.idl` file. These files are created in the `\com\beasys\samples\BankDB` subdirectory.

`Bankapp.ser` `Bankapp.jar`	The Server Descriptor file and Server Java Archive file generated by the `buildjavaserver` command in the make file.

`Bank_c.cpp`
`Bank_c.h`
`Bank_s.cpp`
`Bank_s.h`
	Generated by the `idl` command for the interfaces defined in the `Bank.idl` file.

`BankDB_c.cpp`
`BankDB_c.h`
`BankDB_s.cpp`
`BankDB_s.h`
	Generated by the `idl` command for the interfaces defined in the `BankDB.idl` file.

Table 4-3 Files Generated by the XA Bankapp Sample Application

File	Description
`dbaccess_i.cpp`	Generated from the `DBAccess_i.pc` file by the Oracle Pro*C/C++ compiler.
`BankDataBase.exe`	The WLE server application that implements the `DBAccess` interface.
`TMS_ORA.exe`	The server process for the Transaction Manager service.
`JavaServerXA`	The special version of the JavaServer that uses the same XA switches as the BankDataBase server process.
`stderr`	Generated by the `tmboot` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.err.println` method sends the output to the `stderr` file instead of to the `ULOG` file.
`stdout`	Generated by the `tmboot` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.out.println` method sends the output to the `stdout` file instead of to the `ULOG` file.
`tmsysevt.dat`	Contains filtering and notification rules used by the TMSYSEVT (system event reporting) process. This file is generated by the `tmboot` command.

Starting the ATM Client Application in the XA Bankapp Sample Application

Start the ATM client application by entering the following command:

Note: The following command sets the Java CLASSPATH to include the current directory and the client JAR file (`m3envobj.jar`). The full WLE JAR file (`m3.jar`) can be specified instead of the client JAR file.

Windows NT

```
prompt>java -classpath .;%TUXDIR%\udataobj\java\jdk\m3envobj.jar
-DTOBJADDR=%TOBJADDR% Atm Teller2
```

UNIX

```
ksh prompt>java -classpath .:$TUXDIR/udataobj/java/jdk
/m3envobj.jar -DTOBJADDR=$TOBJADDR Atm Teller2
```

The GUI for the ATM client application appears. Figure 3-2 shows the GUI for the ATM client application.

Stopping the XA Bankapp Sample Application

Before using another sample application, enter the following commands to stop the XA Bankapp sample application and to remove unnecessary files from the work directory:

Windows NT

```
prompt>tmshutdown -y
```

```
prompt>nmake -f makefileX.nt clean
```

UNIX

```
ksh prompt>tmshutdown -y
ksh prompt>make -f makefileX.mk clean
```

Using the ATM Client Application

The ATM client application in the XA Bankapp sample application works as it does in the JDBC Bankapp sample application.

Index