The Java™ 2 Enterprise Edition Developer's Guide

Version 1.2.1
May 2000

901 San Antonio Rd.
Palo Alto, CA 94303 U.S.A.
408-863-3321
http://java.sun.com

Please
Recycle

Contents

Preface

This book describes how to develop and deploy enterprise beans for the Java™ 2 SDK, Enterprise Edition (J2EE™ SDK). The J2EE SDK is the reference implementation provided by Sun Microsystems, Inc. for the J2EE platform, a component-based architecture for creating object-oriented, enterprise-level applications. To create an application, you assemble components written in the Java programming language. The components, which are called enterprise beans, implement business tasks or business entities.

Who Should Use This Book

This manual is intended for programmers interested in developing and deploying J2EE applications. To understand the coding examples in this book you will need a basic knowledge of the Java programming language, SQL, and relational database concepts.

Most J2EE applications will access enterprise beans via Web components. Although Web components are important, this manual concentrates on enterprise beans. For more information on developing Web components, see the home pages for the JavaServer Pages™ (http://java.sun.com/products/jsp/) andJava Servlet (http://java.sun.com/products/servlet/) technologies

This book is not intended for tools vendors. It does not explain how to implement the J2EE architecture, nor does it explain the internals of the J2EE SDK. The Enterprise JavaBeans™ Specifications and Java™ 2 Enterprise Edition Specifications describe the J2EE architecture in detail.

Where to Find the Code Examples

The sample code in this book is in the `doc/guides/ejb/examples` directory of your J2EE SDK installation. For instance, the `ConverterEJB.java` file discussed in the Getting Started chapter is in the `doc/guides/ejb/examples/converter` directory. See Appendix A: Code Examples for a full list of code examples.

How to PrintThis Book

To print this book, follow these steps:

- Ensure that Adobe Acrobat Reader is installed on your system.
- Using your browser, access the PDF version of this book at:

  `http://java.sun.com/j2ee/j2sdkee/devguide1_2_1.pdf`
- Click the printer icon in Adobe Acrobat Reader.

Typographical Conventions

A `typewriter` font is used for keywords within explanatory text or for code examples. An ellipses (. . .) within a code example indicates omission. These code examples will not compile. An *Italic* font is used when defining a new term.

CHAPTER 1

Overview

Enterprise JavaBeans™ (EJB™) technology is part of a larger framework-- the Java™ 2 Platform, Enterprise Edition (J2EE™). This platform is an architecture for developing, deploying, and executing applications in a distributed environment. These applications require system-level services, such as transaction management, security, client connectivity, and database access. The J2EE platform provides these services, allowing you to focus on the business logic in your applications, not the system-level plumbing. You code the business logic in enterprise beans, reusable components that can be accessed by client programs. Enterprise beans run on a J2EE server, which functions as a middle-tier server in a three-tier client/server system.

Benefits of Middle-Tier Servers

A middle-tier server plays a vital role in a three-tier application. It handles requests from clients, shielding them from the complexity involved in dealing with back office systems and databases. The middle-tier server might support a variety of clients, such as Web browsers, Java applications, and hand held devices. The clients handle the user interface. They do not query databases, execute complex business rules, or connect to legacy applications. They let the middle-tier server do these jobs for them transparently.

Figure 1-1 illustrates a three-tier application. Tier 1 is composed of multiple clients, which request services from the middle-tier server in tier 2. The middle-tier server accesses data from the existing systems in tier 3, applies business rules to the data, and returns the results to the clients in tier 1.

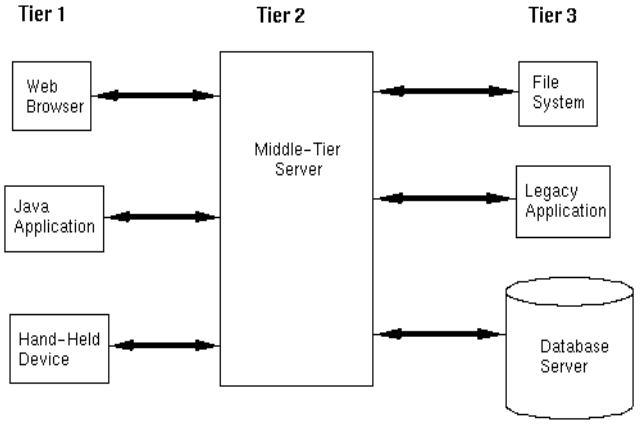

FIGURE 1-1 A Three-Tier Client/Server Application

Middle-tier servers provide business services to clients. For example, a middle- tier server in an online shopping application might provide a variety of services: catalog lookup, order entry, and credit verification.

Middle-tier servers also provide system-level services:

- Remote access to clients and back-office systems
- Session and transaction management
- Security enforcement
- Resource pooling

Because the middle-tier provides these services, the clients can be thin, simple, and rapidly developed. You can integrate new clients with existing applications and databases, protecting your investment in legacy systems.

Middle-tier servers enable you to create large-scale distributed applications for the enterprise. The architecture of the J2EE platform makes it the ideal choice for developing middle-tier servers.

J2EE Architecture

The Java™ 2 SDK, Enterprise Edition (J2EE SDK) is the reference implementation provided by Sun Microsystems, Inc. The following figure shows the major elements of the architecture for the J2EE SDK:

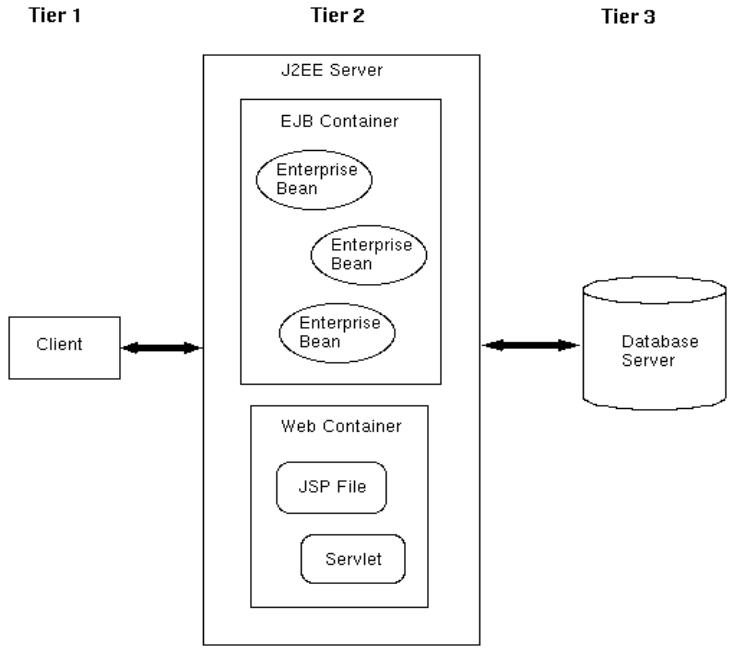

FIGURE 1-2 J2EE Architecture

J2EE Server

The J2EE server provides the following services:

- Naming and Directory - allows programs to locate services and components through the Java Naming and Directory Interface™ (JNDI) API
- Authentication - enforces security by requiring users to log in
- HTTP - enables Web browsers to access servlets and JavaServer Pages™ (JSP) files
- EJB - allows clients to invoke methods on enterprise beans

EJB Container

Enterprise bean instances run within anEJB container. The container is a runtime environment that controls the enterprise beans and provides them with important system-level services. Since you don't have to develop these services yourself, you are free to concentrate on the business methods in the enterprise beans. The container provides the following services to enterprise beans:

- Transaction Management
- Security
- Remote Client Connectivity
- Life Cycle Management
- Database Connection Pooling

Transaction Management

When a client invokes a method in an enterprise bean, the container intervenes in order to manage the transaction. Because the container manages the transaction, you do not have to code transaction boundaries in the enterprise bean. The code required to control distributed transactions can be quite complex. Instead of writing and debugging complex code, you simply declare the enterprise bean's transactional properties in the deployment descriptor file. The container reads the file and handles the enterprise bean's transactions for you.

Security

The container permits only authorized clients to invoke an enterprise bean's methods. Each client belongs to a particular role, and each role is permitted to invoke certain methods. You declare the roles and the methods they may invoke in the enterprise bean's deployment descriptor. Because of this declarative approach, you don't need to code routines that enforce security.

Remote Client Connectivity

The container manages the low-level communications between clients and enterprise beans. After an enterprise bean has been created, a client invokes methods on it as if it were in the same virtual machine.

Life Cycle Management

An enterprise bean passes through several states during its lifetime. The container creates the enterprise bean, moves it between a pool of available instances and the active state, and finally, removes it. Although the client calls methods to create and remove an an enterprise bean, the container performs these tasks behind the scenes.

Database Connection Pooling

A database connection is a costly resource. Obtaining a database connection is time-consuming and the number of connnections may be limited. To alleviate these problems, the container manages a pool of database connections. An enterprise bean can quickly obtain a connection from the pool. After the bean releases the connection, it may be re-used by another bean.

Web Container

The Web container is a runtime environment for JSP files and and servlets. Although these Web components are an important part of a J2EE application, this manual focuses on enterprise beans. For more information on developing Web components, see the home pages for the JavaServer Pages™ (www.java.sun.com/products/jsp/) and Java Servlet (www.java.sun.com/products/servlet/) technologies.

Enterprise Beans

Enterprise beans are server components written in the Java programming language. Enterprise beans contain the business logic for your application. For example, a checkbook client might invoke the `debit` and `credit` methods of an account enterprise bean.

There are two types of enterprise beans: session beans and entity beans.

Session Beans

A session bean represents a client in the J2EE server. A client communicates with the J2EE server by invoking the methods that belong to an enterprise bean. For example, an online shopping client might invoke the `enterOrder` method of its session bean to create an order. A session bean converses with the client, and can be thought of as an extension of the client. Each session bean can have only one client. When the client terminates, its corresponding session bean also terminates. Therefore, a session bean is transient, or non-persistent.

Entity Beans

An entity bean represents a business object in a persistent storage mechanism such as a database. For example, an entity bean could represent a customer, which might be stored as a row in the customer table of a relational database. An entity bean's information does not have to be stored in a relational database. It could be stored in an object database, a legacy application, a file, or some other storage mechanism. The type of storage mechanism depends on the particular implementation of EJB technology. The reference implementation (J2EE SDK) uses a relational database. See the section "Database Access" on page 10 for more information.

The persistence of an entity bean can be managed by either the entity bean itself, or by the EJB container. Bean-managed persistence requires you to write the data access code in the Bean. For example, a customer entity bean would include the SQL commands to access a relational database via JDBC. Container-managed persistence means that the EJB container handles the data access calls automatically.

Comparing Session and Entity Beans

Although both session and entity beans run in an EJB container, they are quite different. The following table contrasts session and entity beans:

TABLE 1-1 Differences Between Session and Entity Beans

	Session Bean	Entity Bean
Purpose	Performs a task for a client.	Represents a business entity object that exists in persistent storage.
Shared Access	May have one client.	May be shared by multiple clients.
Persistence	Not persistent. When the client terminates its session bean is no longer available.	Persistent. Even when the EJB container terminates, the entity state remains in a database.

The flexibility of the EJB architecture allows you to build applications in a variety of ways. The following illustration shows how you might create an online shopping application with both session and entity beans. An HTML form displayed in a Web browser accesses a servlet in a Web container. The servlet is the client of a shopping session bean. When the HTML form needs to find a product or enter an order, it instructs the servlet to call the appropriate business methods in the session bean. The session bean is the client of the order, product, and customer entity beans. Because entity beans are persistent, their state is stored in the database.

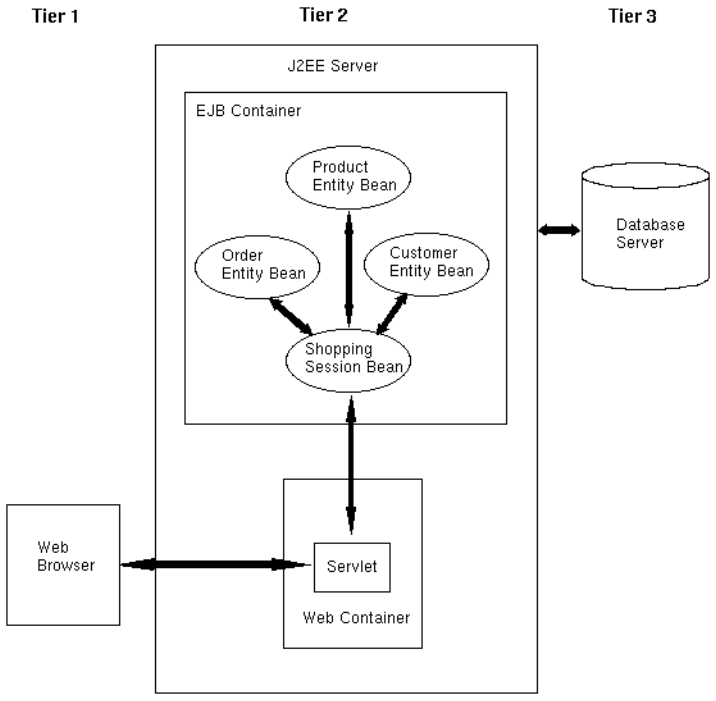

FIGURE 1-3 Using Session and Entity Beans

Java Beans™ Components and Enterprise Beans

JavaBeans components and enterprise beans are not the same. Although both components are written in the Java programming language, they are not interchangeable. JavaBeans components define a convention for making a Java class instance customizable by design tools, allowing the tools to link these customized objects via events. Enterprise beans implement multi-user, transactional services.

Programming Restrictions for Enterprise Beans

Enterprise beans make use of the services provided by the EJB container, such as life-cycle management. To avoid conflicts with these services, enterprise beans are restricted from performing certain operations:

- Managing or synchronizing threads
- Accessing files or directories with the `java.io` package

- Using AWT functionality to display information or to accept information from a keyboard
- Listening on a socket, accepting connections on a socket, or using a socket for multicast
- Setting a socket factory used by `ServerSocket`, `Socket`, or the stream handler factory used by the `URL` class
- Loading a native library

Database Access

The Enterprise JavaBeans specification does not require an implementation to support a particular type of database. Therefore, the databases supported by different J2EE implementations may vary. See the Release Notes for a list of the databases currently supported by the J2EE SDK.

Both session and entity beans can access a database. The type of enterprise bean you choose depends on your application. You might want to include SQL calls in a session bean under the following circumstances:

- The application is relatively simple.
- The data returned by the SQL call will not be used by multiple clients.
- The data does not represent a business entity.

You should probably access a database from an entity bean if any of the following conditions are true:

- More than one client will use the data returned by the database call.
- The data represents a business entity.
- You want to hide the relational model from the session bean.

J2EE Applications

You assemble a J2EE application from three kinds of modules: enterprise beans, Web components, and J2EE application clients. These modules are reusable-- you can build new applications from existing enterprise beans and components. And because the modules are portable, the application they comprise will run on any J2EE server that conforms to the specifications.

Contents of a J2EE Application

Figure 1-4 shows the hierarchy of a J2EE application. A J2EE application may contain one or more enterprise beans, Web components, or J2EE application client components. An enterprise bean is composed of three class files-- the EJB class, the remote interface, and the home interface. (The next chapter discusses these class files in more detail.) A Web component may contain files of the following types: servlet class, JSP, HTML, and GIF. A J2EE application client is a Java application that runs in an environment (container) which allows it to access J2EE services.

Each J2EE application, Web component, enterprise bean, and J2EE application client has a deployment descriptor. (Figure 1-4 abbreviates a deployment descriptor as DD.) A deployment descriptor is an .xml file that describes the component. An EJB deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. Because this information is declarative, it can be changed without requiring modifications to the bean's source code. At run time, the J2EE server reads this information and acts upon the bean accordingly.

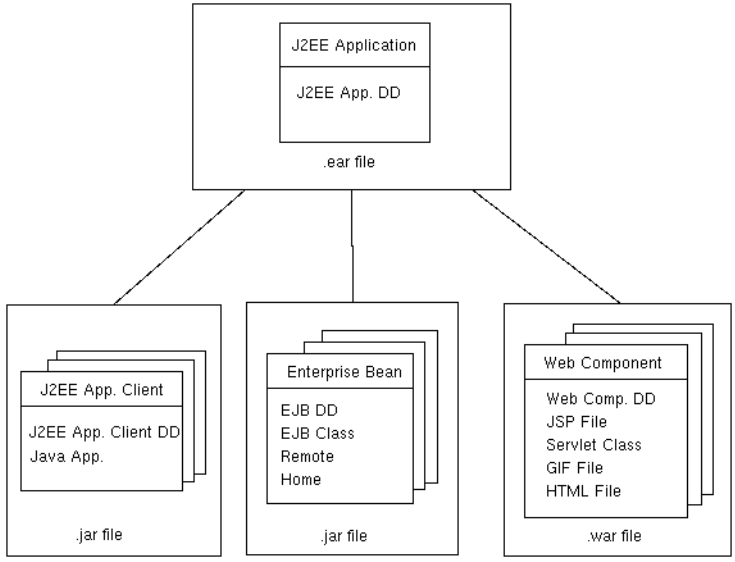

FIGURE 1-4 Contents of a J2EE Application

You bundle each module into a file with a particular format-- a J2EE application in a .ear file, an enterprise bean in an EJB .jar file, a Web component in a .war file, and a J2EE application client in a .jar file. An .ear file, for example, contains an .xml file for its deployment descriptor, and one or more EJB .jar and .war files. An EJB .jar contains its deployment descriptor and the .class files for the enterprise bean. The following table lists the file type of every element residing in a J2EE application.

TABLE 1-2 Files Used in a J2EE Application

Element	File Type
J2EE Application	.ear
J2EE Application Deployment Descriptor	.xml
Enterprise Bean	ejb .jar
EJB Deployment Descriptor	.xml
EJB Class	.class
Remote Interface	.class
Home Interface	.class
Web Component	.war
Web Component Deployment Descriptor	.xml
JSP File	.jsp
Servlet Class	.class
GIF File	.gif
HTML File	.html
J2EE Application Client	.jar
J2EE Application Client Deployment Descriptor	.xml
Java Application	.class

Development Phases of J2EE Applications

As a J2EE application evolves, it passes through these development phases:

- Enterprise Bean Creation
- Web Component Creation

- J2EE Application Client Creation
- J2EE Application Assembly
- J2EE Application Deployment

In a large organization, each phase might be executed by different individuals or teams. This division of labor works because each of the earlier phases outputs a portable file that is the input for a subsequent phase. For example, in the Enterprise Bean Creation phase, a software developer delivers EJB .jar files. In the J2EE Application phase, another developer combines these EJB .jar files into a J2EE application and saves it in an .ear file. In the final phase, J2EE Application Deployment, a system administrator at the customer site uses the .ear file to install the J2EE application into a J2EE server. Figure 1-5 illustrates these last two phases.

The different phases are not always executed by different people. If you work for a small company, for example, or if you are protyping a sample application, you might peform the tasks in every phase.

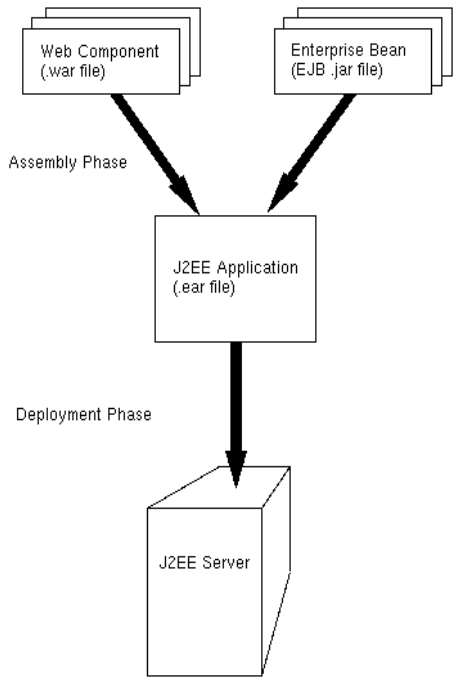

FIGURE 1-5 J2EE Application Assembly and Deployment

The sections that follow summarize the development phases for J2EE applications. Because a J2EE application is not required to have every type of module, only one of the first three phases is required. The final two phases are necessary. Since this

manual focuses on enterprise beans, it does not discuss the Web Component Creation phase. The next chapter in this manual, Getting Started, shows you how to build and deploy a sample J2EE application that contains an enterprise bean.

Enterprise Bean Creation

Person: software developer

Tasks:

- Codes and compiles the Java source code needed by the enterprise bean
- Specifies the deployment descriptor for the enterprise bean
- Bundles the .class files and deployment descriptor into an EJB .jar file

Deliverable: the EJB .jar file containing the enterprise bean

Web Component Creation

Persons: Web designer (JavaServer Pages components), software developer (servlets)

Tasks:

- Codes and compiles Java source code for the servlet
- Writes .jsp and .html files
- Specifies the deployment descriptor for the Web component
- Bundles the .class, .jsp, .html, and deployment descriptor files into the .war file

Deliverable: the .war file containing the Web component

J2EE Application Client Creation

Person: software developer

Tasks:

- Codes and compiles the Java source code needed by the client
- Specifies the deployment descriptor for the client
- Bundles the .class files and deployment descriptor into the .jar file for the client.

Deliverable: the .jar file containing the J2EE application client

J2EE Application Assembly

Person: software developer

Tasks:

- Assembles enterprise beans (EJB .jar) and Web components (.war) created in the previous phases into a J2EE application (.ear)
- Specifies the deployment descriptor for the J2EE application

Deliverable: the .ear file containing the J2EE application

J2EE Application Deployment

Person: system administrator

Tasks:

- Adds the J2EE application (.ear) created in the preceding phase to the J2EE server
- Configures the J2EE application for the operational environment by modifying the deployment descriptor of the J2EE application
- Deploys (installs) the J2EE application (.ear) into the J2EE server

Deliverable: an installed and configured J2EE application

CHAPTER 2

Getting Started

This chapter shows you how to develop, deploy, and run a simple client-server application that uses an enterprise bean. The client is a stand-alone Java™ application named `ConverterClient`. It performs simple currency conversions by calling the methods of an enterprise bean named `ConverterEJB`. You can find the source code in the `doc/guides/ejb/examples/converter` subdirectory of your Java™ 2 SDK, Enterprise Edition (J2EE SDK) installation.

To implement the sample application in this chapter you perform these tasks:

- Coding the Enterprise Bean
- Creating the J2EE Application
- Packaging the Enterprise Bean
- Deploying the J2EE Application
- Building the Client
- Running the Client

Coding the Enterprise Bean

Every enterprise bean requires the following code:

- Remote interface
- Home interface
- Enterprise bean class

Coding the Remote Interface

A remote interface defines the business methods that a client may call. The business methods are implemented in the enterprise bean code. The source code for the `Converter` remote interface follows.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Converter extends EJBObject {

   public double dollarToYen(double dollars) throws RemoteException;
   public double yenToEuro(double yen) throws RemoteException;
}
```

Coding the Home Interface

A home interface defines the methods that allow a client to create, find, or remove an enterprise bean. The `ConverterHome` interface contains a single `create` method, which returns an object of the remote interface type. Here is the source code for the `ConverterHome` interface:

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface ConverterHome extends EJBHome {

    Converter create() throws RemoteException, CreateException;
}
```

Coding the Enterprise Bean Class

The enterprise bean in our example is a stateless session bean called `ConverterEJB`. This bean implements the two business methods, `dollarToYen` and `yenToEuro`, that the `Converter` remote interface defines. The source code for the `ConverterEJB` class follows.

```
import java.rmi.RemoteException;
```

```
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class ConverterEJB implements SessionBean {

   public double dollarToYen(double dollars) {

      return dollars * 121.6000;
   }

   public double yenToEuro(double yen) {

      return yen * 0.0077;
   }

   public ConverterEJB() {}
   public void ejbCreate() {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}
}
```

Compiling the Enterprise Bean's Source Code

Now you are ready to compile the remote interface (`Converter.java`), home interface (`ConverterHome.java`) , and the enterprise bean class (`ConverterEJB.java`).

UNIX:

1. In the following script, `compileEJB.sh`, change <installation-location> to the directory in which you installed the J2EE SDK.

```
#!/bin/sh
```

```
J2EE_HOME=<installation-location>
CPATH=.:$J2EE_HOME/lib/j2ee.jar
```

```
javac -classpath "$CPATH" ConverterEJB.java ConverterHome.java Converter.java
```

2. Run the compileEJB.sh script.

Windows:

1. In the following script, `compileEJB.bat`, change <installation-location> to the directory in which you installed the J2EE SDK.

```
set J2EE_HOME=<installation-location>
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar
```

```
javac -classpath %CPATH% ConverterEJB.java ConverterHome.java Converter.java
```

2. Run the `compileEJB.bat` script.

Creating the J2EE Application

You cannot deploy an enterprise bean directly into an J2EE server. Instead, you add the bean to a J2EE application, which you then deploy. In this section, you will create a new J2EE application called ConverterApp and store it in the file named `ConverterApp.ear`.

1. At the command line prompt, start the J2EE server:

```
j2ee -verbose
```

(To stop the server, type `j2ee -stop`.)

2. In another terminal window, run the Application Deployment Tool:

```
deploytool
```

(To access the tool's context-sensitive help, press f1.)

3. Create a new J2EE application.

a. In the Application Deployment Tool, select the File menu.

b. From the File menu choose New Application.

c. Click Browse.

d. In the file chooser, locate the directory where you want to place the .ear file that contains the J2EE application.

e. In the File name field enter `ConverterApp.ear`.

f. Click New Application.

g. Click OK.

Packaging the Enterprise Bean

In this section you will run the New Enterprise Bean Wizard of the Application Deployment Tool to perform these tasks:

- Create the bean's deployment descriptor.
- Package the deployment descriptor and the bean's classes in an EJB .jar file.
- Insert the EJB .jar file into the application's `ConverterApp.ear` file.

To start the New Enterprise Bean Wizard, from the File menu choose New Enterprise Bean. The wizard displays the following dialog boxes.

Introduction Dialog Box:

a. Read this explanatory text for an overview of the wizard's features.

b. Click Next.

EJB JAR Dialog Box:

a. In the combo box labelled "Enterprise Bean will go in," select ConverterApp.

b. In the JAR Display Name field enter ConverterJAR. Representing the EJB .jar file that contains the bean, this name will be displayed in the tree view.

c. Click the Add button next to the Contents text area.

d. In the Add Contents to JAR dialog box, choose the directory containing the .class files. (This directory is `$J2EE_HOME/doc/guides/ejb/examples/converter`, where `$J2EE_HOME` is the location of your J2EE SDK installation.) You may either type the directory name in the Root Directory field or locate it by clicking Browse.

e. Select each of the following classes from the text area and click Add: `Converter.class`, `ConverterEJB.class`, and `ConverterHome.class`.

f. Click OK.

g. Click Next.

General Dialog Box:

a. In the Enterprise Bean Class combo box, select ConverterEJB.

b. In the Home Interface combo box, select ConverterHome.

c. In the Remote Interface combo box, select Converter.

d. Select the Session radio button.

e. Select the Stateless radio button.

f. In the Enterprise Bean Display Name field, enter ConverterBean. This name will represent the enterprise bean in the tree view.

g. Click Next.

Environment Entries Dialog Box:

Because you may skip the remaining dialog boxes, click Finish.

Deploying the J2EE Application

Now that the J2EE application contains an enterprise bean, it is ready for deployment.

1. Specify the JNDI name of the enterprise bean.

   a. In the Application Deployment Tool, select ConverterApp in the tree view.

   b. Select the JNDI Names tab.

   c. In the JNDI Names field, enter `MyConverter` and press Return. The client will use this name to locate the home interface. (See a later section, “Locate the Home Interface” on page 23.)

2. Deploy the J2EE application.

   a. From the Tools menu, choose Deploy Application.

   b. In the first dialog box, verify that the Target Server selection is either “localhost” or the name of the host running the J2EE server.

   c. Select the checkbox labelled “Return Client Jar.”

d. In the text field that appears, enter the full path name for the file `ConverterAppClient.jar`. This file will reside in the `doc/guides/ejb/examples/converter` subdirectory of your J2EE SDK installation.

e. Click Next.

f. In the second dialog box, verify that the JNDI name for the ConverterBean is MyConverter.

g. Click Next.

h. In the third dialog box, click Finish.

i. In the Deployment Progress dialog box, click OK when the deployment completes.

Building the Client

The `ConverterClient` program is a stand-alone Java application. To create `ConverterClient` you perform these steps:

1. Coding the Client

2. Compiling the Client's Code

Coding the Client

The `ConverterClient.java` source code illustrates the basic tasks performed by the client of an enterprise bean:

- Locate the Home Interface
- Create an Enterprise Bean Instance
- Invoke a Business Method

Locate the Home Interface

The `ConverterHome` interface defines life cycle methods such as `create`. Before the `ConverterClient` can invoke the `create` method, it must instantiate an object whose type is `ConverterHome`. This is a three-step process:

1. Create a JNDI naming context.

```
Context initial = new InitialContext();
```

2. Retrieve the object bound to the name `MyConverter`. (This is the JNDI name you specified in the section, “Deploying the J2EE Application” on page 22.)

```
java.lang.Object objref = initial.lookup(“MyConverter”);
```

3. Narrow the reference to a `ConverterHome` object.

```
ConverterHome home =
           (ConverterHome) PortableRemoteObject.narrow(objref,
                                              ConverterHome.class);
```

Create an Enterprise Bean Instance

To create the `ConverterEJB` class, the client invokes the `create` method on the `ConverterHome` object. The `create` method returns an object whose type is `Converter`. The remote `Converter` interface defines the business methods in `ConverterEJB` that the client may call. When the client invokes the `create` method, the EJB container instantiates `ConverterEJB`, and then invokes the `ConverterEJB.ejbCreate` method.

```
Converter currencyConverter = home.create();
```

Invoke a Business Method

Calling a business method is easy. You simply invoke the method on the `Converter` object. The EJB container will invoke the corresponding method on the `ConverterEJB` instance that is running on the server. The client invokes the `dollarToYen` business method in the following line of code.

```
double amount = currencyConverter.dollarToYen(100.00);
```

ConverterClient Source Code

The full source code for the `ConverterClient` program follows.

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

import Converter;
import ConverterHome;

public class ConverterClient {
```

```
    public static void main(String[] args) {
        try {
            Context initial = new InitialContext();
            Object objref = initial.lookup("MyConverter");

            ConverterHome home =
                (ConverterHome)PortableRemoteObject.narrow(objref,
                                                ConverterHome.class);

            Converter currencyConverter = home.create();

            double amount = currencyConverter.dollarToYen(100.00);
            System.out.println(String.valueOf(amount));
            amount = currencyConverter.yenToEuro(100.00);
            System.out.println(String.valueOf(amount));

            currencyConverter.remove();

        } catch (Exception ex) {
            System.err.println("Caught an unexpected exception!");
            ex.printStackTrace();
        }
    }
}
```

Compiling the Client's Code

UNIX:

1. In the following script, `compileClient.sh`, change <installation-location> to the directory in which you installed the J2EE SDK.

```
#!/bin/sh

J2EE_HOME=<installation-location>
CPATH=.:$J2EE_HOME/lib/j2ee.jar
```

```
javac  -classpath "$CPATH" ConverterClient.java
```

2. Run the `compileClient.sh` script.

Windows:

1. In the following script, `compileClient.bat`, change <installation-location> to the directory in which you installed the J2EE SDK.

```
set J2EE_HOME=<installation-location>
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar

javac  -classpath %CPATH% ConverterClient.java
```

2. Run the `compileClient.bat` script.

Running the Client

To run the client you need the the `ConverterAppClient.jar` file. This jar file contains stub classes that allow the client to communicate with the enterprise bean instance that is running in the EJB container. You created the `ConverterAppClient.jar` file in the section, "Deploying the J2EE Application" on page 22.

UNIX:

1. In the following script, `testClient.sh`, change <installation-location> to the directory in which you installed the J2EE SDK.

```
#!/bin/sh

J2EE_HOME=<installation-location>
CPATH=$J2EE_HOME/lib/j2ee.jar:ConverterAppClient.jar:.

java  -classpath "$CPATH" ConverterClient
```

2. Run the `testClient.sh` script.

Windows:

1. In the following script, `testClient.bat`, change <installation-location> to the directory in which you installed the J2EE SDK.

```
set J2EE_HOME=<installation-location>
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar;ConverterAppClient.jar

java -classpath "%CPATH%" ConverterClient
```

2. Run the `testClient.bat` script.

Solutions to Common Problems

When running the `ConverterClient`, you may encounter one of the following errors.

1. `java.lang.ClassCastException`

   The program could not locate the classes in the `ConverterAppClient.jar` file. Make sure that this file exists. You should have specified the `ConverterAppClient.jar` file in the section, "Deploying the J2EE Application" on page 22.

2. `java.lang.NoClassDefFoundError: ConverterClient`

   The `java` launcher could not locate the `ConverterClient.class`. Verify that you compiled `ConverterClient.java` in the section, "Compiling the Client's Code" on page 25.

3. `java.lang.NoClassDefFoundError: javax/naming/Context`

   The program could not locate a class it needed in the `lib/j2ee.jar` file. Make sure that you correctly defined the J2EE_HOME environment variable in the `testClient.sh` (UNIX) or `testClient.bat` (Windows) script.

4. `javax.naming.NameNotFoundException: Lookup of name MyConverter failed.`

   The J2EE server could not locate the enterprise bean whose JNDI name is `MyConverter`. Either you did not deploy the enterprise bean, or you specified the wrong JNDI name in the section, "Deploying the J2EE Application" on page 22.

5. `javax.naming.NamingException: Error accessing repository: Cannot connect to ORB at . . . .`

   The J2EE server is not running. See step 1 in the section, "Creating the J2EE Application" on page 20.

Modifying the J2EE Application

To modify a .class file in an enterprise bean, you change the source code, recompile it, and redeploy the application. For example, suppose that you want to change the exchange rate in the `dollarToYen` business method of the `ConverterEJB` class:

1. Edit the `ConverterEJB.java` source file.

2. Recompile `ConverterEJB.java`.

3. In the Application Deployment Tool, from the Tools menu select Update and Redeploy Application. (The Application Deployment Tool will automatically replace the old .class file in the `ConverterApp` with the new one.)

To add another .class file to the `ConverterJAR` (EJB .jar) of the application, you would perform these tasks:

1. Select `ConverterJAR` in the tree view.

2. Select the General tab.

3. Click the Add button to the right of the Contents field.

4. In the Add Files to JAR dialog box, locate the .class file and click Add.

5. From the Tools menu, select Update and Redeploy Application.

To modify a deployment setting of the `ConverterApp`, you edit the appropriate field in the inspector panel of the Application Deployment Tool and redeploy the application. For example, to change the JNDI name of the `ConverterBean` from `ATypo` to `MyConverter`, you would follow these steps:

1. In the Application Deployment Tool, select the `ConverterApp` in the tree view.

2. Select the JNDI Names tab.

3. In the JNDI Name field, enter `MyConverter`.

4. From the Tools menu, select Update and Redeploy Application.

CHAPTER 3

Session Beans

A session bean represents a single client inside the J2EE server. The client accesses remote services by invoking the session bean's methods. The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared-- it may have just one client, in the same way that an interactive session may have just one user. Like an interactive session, a session bean is not persistent. When the client terminates, its session bean appears to terminate and is no longer associated with the client.

Session beans are powerful because they extend the reach of your clients into remote servers-- yet they're easy to build. The following section shows you how to construct a simple session bean.

A Session Bean Example

The session bean in this section represents the shopping cart in an online book store. The bean's client may add a book to the cart, remove a book, or retrieve the cart's contents. To construct a session bean, you need the following code:

- Session Bean Class (`CartEJB.java`)
- Home Interface (`CartHome.java`)
- Remote Interface (`Cart.java`)

The preceding files are required for any enterprise bean. To meet the needs of a specific application, an enterprise bean may also need some helper classes. The CartEJB session bean uses two helper classes, `BookException` and `IdVerifier`, which are discussed in the section, "Helper Classes" on page 36 .)

The source code for all of these files are in the `doc/guides/ejb/examples/cart` directory, along with `CartClient.java`, the code for the client program.

Session Bean Class

The session bean class for this example is called `CartEJB`. Like any session bean, the `CartEJB` class must meet these requirements:

- It implements the `SessionBean` interface.
- The class is defined as `public`.
- The class cannot be defined as `abstract` or `final`.
- It implements one or more `ejbCreate` methods.
- It implements the business methods.
- It contains a `public` constructor with no parameters.
- It must not define the `finalize` method.

The source code for `CartEJB` follows:

```
import java.util.*;
import javax.ejb.*;

public class CartEJB implements SessionBean {

   String customerName;
   String customerId;
   Vector contents;

   public void ejbCreate(String person) throws CreateException {

      if (person == null) {
        throw new CreateException("Null person not allowed.");
      }
      else {
         customerName = person;
      }

      customerId = "0";
      contents = new Vector();
```

```
}

public void ejbCreate(String person, String id)
  throws CreateException {

   if (person == null) {
     throw new CreateException("Null person not allowed.");
   }
   else {
      customerName = person;
   }

   IdVerifier idChecker = new IdVerifier();
   if (idChecker.validate(id)) {
      customerId = id;
   }
   else {
      throw new CreateException("Invalid id: " + id);
   }

   contents = new Vector();
}

public void addBook(String title) {

   contents.addElement(title);
}

public void removeBook(String title) throws BookException {

   boolean result = contents.removeElement(title);
   if (result == false) {
      throw new BookException(title + " not in cart.");
   }
}
```

```
    public Vector getContents() {
        return contents;
    }

    public CartEJB() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext sc) {}

}
```

The SessionBean Interface

The `SessionBean` interface extends the `EnterpriseBean` interface, which in turn extends the `Serializable` interface. The `SessionBean` interface declares the `ejbRemove`, `ejbActivate`, `ejbPassivate`, and `setSessionContext` methods. The `CartEJB` class doesn't use these methods, but it must implement them because they're declared in the `SessionBean` interface. Consequently, these methods are empty in the `CartEJB` class. Later sections explain when you might use these methods.

The ejbCreate Methods

Because an enterprise bean runs inside an EJB container, a client cannot directly instantiate the bean. Only the EJB container can instantiate an enterprise bean. During instantiation, the example program performs these steps:

1. The client invokes a `create` method on the home object:

```
Cart shoppingCart = home.create("Duke DeEarl","123");
```

2. The EJB container instantiates the enterprise bean.

3. The EJB container invokes the appropriate `ejbCreate` method in `CartEJB`:

```
    public void ejbCreate(String person, String id)
      throws CreateException {

      if (person == null) {
        throw new CreateException("Null person not allowed.");
      }
```

```
        else {
            customerName = person;
        }

        IdVerifier idChecker = new IdVerifier();
        if (idChecker.validate(id)) {
            customerId = id;
        }
        else {
            throw new CreateException("Invalid id: " + id);
        }

        contents = new Vector();
    }
```

Typically, an `ejbCreate` method initializes the state of the enterprise bean. The preceding `ejbCreate` method, for example, initializes the `customerName` and `customerId` variables with the arguments passed by the `create` method.

An enterprise bean may have one or more `ejbCreate` methods. The signatures of the methods meet the following requirements:

- The access control modifier must be `public`.
- The return type must be `void`.
- The arguments must be legal types for Java RMI.
- The modifier cannot be `static` or `final`.

The throws clause may include the `javax.ejb.CreateException` and other exceptions that are specific to your application. The `ejbCreate` method usually throws a `CreateException` if an input parameter is invalid.

Business Methods

The primary purpose of a session bean is to run business tasks for the client. The client invokes business methods on the remote object reference that is returned by the `create` method. From the client's perspective, the business methods appear to run locally, but they actually run remotely in the session bean. The following code snippet shows how the `CartClient` program invokes the business methods:

```
Cart shoppingCart = home.create(“Duke DeEarl”, “123”);
. . .
shoppingCart.addBook(“The Martian Chronicles”);
```

```
shoppingCart.removeBook(“Alice In Wonderland”);
bookList = shoppingCart.getContents();
```

The `CartEJB` class implements the business methods in the following code:

```
public void addBook(String title) {

   contents.addElement(new String(title));
}

public void removeBook(String title) throws BookException {

   boolean result = contents.removeElement(title);
   if (result == false) {
      throw new BookException(title + " not in cart.");
   }
}

public Vector getContents() {
   return contents;
}
```

The signature of a business method must conform to these rules:

- The method name must not conflict with one defined by the EJB architecture. For example, you cannot call a business method `ejbCreate` or `ejbActivate`.
- The access control modifier must be `public`.
- The arguments and return types must be legal types for Java RMI.
- The modifier must not be `static` or `final`.

The throws clause may include exceptions that you define for your application. The `removeBook` method, for example, throws the `BookException` if the book is not in the cart.

To indicate a system-level problem, such as the inability to connect to a database, a business method should throw the `javax.ejb.EJBException`. When a business method throws an `EJBException`, the container wraps it in a `RemoteException`, which is caught by the client. The container will not wrap application exceptions such as `BookException`. Because `EJBException` is a subclass of `RuntimeException`, you do not need to include it in the throws clause of the business method.

Home Interface

A home interface extends the `EJBHome` interface. The purpose of the home interface is to define the `create` methods that a client may invoke. The `CartClient` program, for example, invokes this `create` method:

```
Cart shoppingCart = home.create("Duke DeEarl", "123");
```

Every `create` method in the home interface corresponds to an `ejbCreate` method in the bean class. The signatures of the `ejbCreate` methods in the `CartEJB` class follow:

```
public void ejbCreate(String person) throws CreateException
. . .
public void ejbCreate(String person, String id)
   throws CreateException
```

Compare the `ejbCreate` signatures with those of the `create` methods in the `CartHome` interface:

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface CartHome extends EJBHome {
    Cart create(String person) throws RemoteException,
                                      CreateException;
    Cart create(String person, String id) throws RemoteException,
                                                 CreateException;
}
```

The signatures of the `ejbCreate` and `create` methods are similar, but differ in important ways. The rules for defining the signatures of the `create` methods of a home interface follow:

- The number and types of argments in a `create` method must match those of its corresponding `ejbCreate` method.
- The arguments and return type of the `create` method must be valid RMI types.
- A `create` method returns the remote interface type of the enterprise bean. (But an `ejbCreate` method returns `void`.)
- The throws clause of the `create` method must include the `java.rmi.RemoteException` and the `javax.ejb.CreateException`.

Remote Interface

The remote interface, which extends `javax.ejb.EJBObject`, defines the business methods that a client may invoke. Here is the source code for the `Cart` remote interface :

```
import java.util.*;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Cart extends EJBObject {

   public void addBook(String title) throws RemoteException;
   public void removeBook(String title) throws BookException,
                                               RemoteException;
   public Vector getContents() throws RemoteException;
}
```

The method definitions in a remote interface must follow these rules:

- Each method in the remote interface must match a method implemented in the enterprise bean class.
- The signatures of the methods in the remote interface must be identical to the signatures of the corresponding methods in the enterprise bean class.
- The arguments and return values must be valid RMI types.
- The throws clause must include the `java.rmi.RemoteException.`

Helper Classes

The `CartEJB` bean has two helper classes: `BookException` and `IdVerifier`. The `BookException` is thrown by the `removeBook` method and the `IdVerifier` validates the `customerId` in one of the `ejbCreate` methods. Helper classes should reside in the EJB .jar file that contains the enterprise bean class

State Management Modes

When you specify the deployment descriptor of a session bean, you must choose between two state management modes: stateful or stateless.

Stateful Session Beans

The `CartEJB` example (see “Session Bean Class” on page 30) has three instance variables: `customerName`, `customerId`, and `contents`. These variables represent the conversational state of the shopping cart application. Because the `CartEJB` contains a conversational state, it is called a stateful session bean.

The state is retained for the duration of the client-bean session. When the client removes the bean, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean is over there is no need to retain the state.

Stateless Session Beans

A stateless session bean does not maintain a conversational state for a particular client. When a client invokes the method of a stateless bean, the beans’s instance variables may contain a state, but only for the duration of the invocation. When the method is finished, the state is no longer retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.

Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

At times, the EJB container may write a stateful session bean out to secondary storage. However, stateless session beans are never written out to secondary storage. Therefore, stateless beans may offer better performance than stateful beans.

The home interface of a stateless session bean must have a single `create` method with no arguments. The session bean class must contain one `ejbCreate` method, also without arguments. (The arguments are only needed by stateful session beans, which use them to initialize their states.)

Choosing Between Stateful and Stateless Session Beans

You should consider using a stateful session bean if any of the following conditions are true:

- The bean’s state must be initialized when it is created.
- The bean needs to hold information about the client across method invocations.

- The client is an interactive application.

Since the primary goal of a session bean is to represent a client in the J2EE server, most of your session beans will be stateful. However, sometimes you may want to use stateless session beans:

- The bean performs a task that is not tailored to the needs of a particular client. For example, you might use a stateless session bean to fetch from a database a commonly used set of data.
- The bean doesn't need to hold information about the client across method invocations.

The Life Cycle of a Session Bean

A session bean goes through various stages during its lifetime, or life cycle. The life cycle is managed by the EJB container, not by your applications. Although your applications cannot explicity manage a bean's life cycle, you'll find the information in the following sections useful when you need to manage resources such as database connections. (See the Database Connections chapter for details.)

The Stateful Session Bean Life Cycle

Figure 3-1 illustrates the stages that a session bean passes through during its lifetime. The client initiates the life cycle by invoking the `create` method.The EJB container instantiates the bean and then invokes the `setSessionContext` and `ejbCreate` methods in the session bean. The bean is now ready to have its business methods invoked.

While in the ready stage, the EJB container may decide to deactivate, or *passivate*, the bean by moving it from memory to secondary storage. (Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation.) The EJB container invokes the bean's `ejbPassivate` method immediately before passivating it. If a client invokes a business method on the bean while it is in the passive stage, the EJB container activates the bean, moving it back to the ready stage, and then calls the bean's `ejbActivate` method.

At the end of the life cycle, the client invokes the `remove` method and the EJB container calls the bean's `ejbRemove` method. The bean's instance is ready for garbage collection.

Your code controls the invocation of only two life cycle methods-- the `create` and `remove` methods in the client. All other methods in Figure 3-1 are invoked by the EJB container. The `ejbCreate` method, for example, is inside the bean class,

allowing you to perform certain operations right after the bean is instantiated. For instance, you may wish to connect to a database in the `ejbCreate` method. (See the Database Connections chapter for more information.)

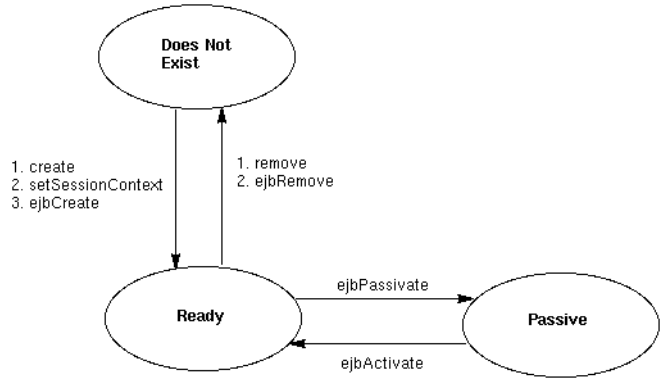

FIGURE 3-1 Life Cycle of a Stateful Session Bean

The Stateless Session Bean Life Cycle

Because a stateless session bean is never passivated, its life cycle has just two stages: non-existent and ready for the invocation of business methods. Figure 3-2 illustrates the stages of a stateless session bean.

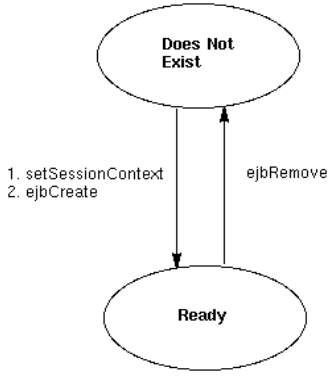

FIGURE 3-2 Life Cycle of a Stateless Session Bean

Comparing Session Beans

A client can determine if the object references of two stateful session beans are identical by invoking the `isIdentical` method:

```
bookCart = home.create(“Bill Shakespeare”);
videoCart = home.create(“Lefty Lee”);
. . .
if (bookCart.isIdentical(bookCart)) {
   // true . . . }
if (bookCart.isIdentical(videoCart)) {
   // false . . . }
```

Because stateless session beans have the same object identity, the `isIdentical` method always returns true when used to compare two such beans.

Passing a Session Bean’s Object Reference

Suppose that your session bean needs to pass a reference to itself to another bean. You might want to pass the reference, for example, so that the second bean can call your session bean’s methods. You can’t pass the `this` reference because the session bean is not a remote object. Instead, your bean must pass an object reference for the instance. It gets the reference to itself by calling the `getEJBObject` method of the `SessionContext` interface. This interface provides a session bean with access to the instance context maintained by the EJB container. Typically, the bean saves the context in the `setSessionContext` method. The following code fragment shows how you might use these methods.

```
public class WagonEJB implements SessionBean {

   SessionContext context;
   . . .
   public void setSessionContext(SessionContext sc) {
      this.context = sc;
   }
   . . .
```

```
public void passItOn(Basket basket) {
. . .
   basket.copyItems(context.getEJBObject());
}
. . .
```

Accessing Environment Entries

Note: The material described in this section applies to both session and entity beans.

Stored in an enterprise bean's deployment descriptor, an environment entry is a name-value pair that allows you to customize the bean's business logic without changing its source code. An enterprise bean that calculates discounts, for example, might have an environment entry named "Discount Percent." Before deploying the bean's application, you could assign "Discount Percent" a value of .05 on the Environment tabbed pane of the Application Deployment Tool. When you run the application, the enterprise bean fetches the .05 value from its environment.

In the following code example, the `applyDiscount` method uses environment entries to calculate a discount based on the purchase amount. First, the method locates the environment naming context by invoking `lookup` with the "java:comp/env" parameter. Then it calls `lookup` on the environment to get the values for the "Discount Level" and "Discount Percent" names. If you had assigned a value of .05 to the "Discount Percent" name in the Application Deployment Tool, the code will assign .05 to the `discountPercent` variable. The full source code for the enterprise bean that contains the `applyDiscount` method is in the `CheckerEJB` class.

```
public double applyDiscount(double amount) {

      try {

         double discount;

         Context initial = new InitialContext();
         Context environment =
           (Context)initial.lookup("java:comp/env");

         Double discountLevel =
```

```
            (Double)environment.lookup("Discount Level");
         Double discountPercent =
            (Double)environment.lookup("Discount Percent");

         if (amount >= discountLevel.doubleValue()) {
            discount = discountPercent.doubleValue();
         }
         else {
            discount = 0.00;
         }

         return amount * (1.00 - discount);

      } catch (NamingException ex) {
           throw new EJBException("NamingException: " +
              ex.getMessage());
      }
   }
```

CHAPTER 4

Entity Beans

An entity bean represents an entity kept in a persistent storage mechanism, usually a database. A business application, for example, might use a database to store business entity objects such as accounts, customers, orders, and products. Inside the J2EE server, this application would represent the business entity objects with entity beans.

Characteristics of Entity Beans

Entity beans differ from session beans in several ways. Entity beans are persistent, allow shared access, and have primary keys.

Persistence

Because the state of an entity bean is saved in a storage mechanism, it is persistent. Persistence means that the entity bean exists beyond the lifetime of the application or the J2EE server process. If you've worked with databases, you're familiar with persistent data. The data in a database is persistent because it still exists even after you shut down the database server or the applications it services.

There are two types of persistence: bean-managed and container-managed. You declare the persistence type with the Application Deployment Tool, which stores the information in the entity bean's deployment descriptor.

With bean-managed persistence, the entity bean code that you write contains the calls that access the database. The `ejbCreate` method, for example, will issue the SQL `insert` statement. You are responsible for coding the `insert` statement and any other necessary SQL calls.

If the container manages an entity bean's persistence, it automatically generates the necessary database access calls. For example, when a client creates an entity bean, the container generates a SQL insert statement. The code that you write for the entity bean does not include any SQL calls. The container also synchronizes the entity bean's instance variables with the data in the underlying database. These instance variables are often referred to as *container-mananged fields*. You declare the container-managed fields with the Application Deployment Tool, which enters the list of fields in the deployment descriptor.

Container-managed persistence has two advantages over bean-managed persistence. First, entity beans with container-managed persistence require less code. Second, because the beans don't contain the database access calls, the code is independent of any particular data store, such as a relational database. However, container-managed persistence has several limitations. See the Release Notes for a complete list of limitations.

Shared Access

Entity beans may be shared by multiple clients. Because the clients might want to change the same data, it's important that entity beans work within transactions. Typically, the EJB container provides transaction management. You specify the transaction attributes in the bean's deployment descriptor. You do not have to code the transaction boundaries in the bean-- the container marks the boundaries for you. See the chapter on Transactions for more information.

Primary Key

Each entity bean has a unique object identifier. A customer entity bean, for example, might be identified by a customer number. The unique identifier, or primary key, enables the client to locate a particular entity bean. For more information, see the section, "Primary Key Class" on page 66.

A Bean-Managed Persistence Example

The entity bean illustrated in this section represents a simple bank account. The state of the entity bean is stored in the `ACCOUNT` table of a relational database. The `ACCOUNT` table was created by the following SQL statement:

```
CREATE TABLE account
   (id        VARCHAR(3) CONSTRAINT pk_account PRIMARY KEY,
```

```
    firstname VARCHAR(24),
    lastname  VARCHAR(24),
    balance   DECIMAL(10,2));
```

To write an entity bean, you must provide the following code:

- Entity Bean Class()
- Home Interface (`AccountHome.java`)
- Remote Interface (`Account.java`)

This example also makes use of the following classes:

- A helper class named `InsufficientBalanceException`
- A client class called `AccountClient.java`

The source code for all of these classes are in the `doc/guides/ejb/examples/account` directory.

Entity Bean Class

The sample entity bean class is called `AccountEJB`. As you look through its code, note that it meets the requirements of every entity bean:

- It implements the `EntityBean` interface.
- The class is defined as `public`.
- The class cannot be defined as `abstract` or `final`.
- It implements zero or more `ejbCreate` and `ejbPostCreate` methods.
- It implements the finder methods (only for bean-managed persistence).
- It implements the business methods.
- It contains an empty constructor.
- It does not implement the `finalize` method.

The EntityBean Interface

The `EntityBean` interface extends the `EnterpriseBean` interface, which extends the `Serializable` interface. The `EntityBean` interface declares a number of methods, such as `ejbActivate` and `ejbLoad`, which you must implement in your entity bean class. These methods are discussed later sections.

The ejbCreate Method

When the client invokes a `create` method, the EJB container invokes the corresponding `ejbCreate` method. Typically, an `ejbCreate` method in an entity bean performs the following tasks:

- Inserts the entity state into the database.
- Initializes the instance variables.
- Returns the primary key.

The `ejbCreate` method of `AccountEJB` inserts the entity state into the database by invoking the private `insertRow` method, which issues the SQL `insert` statement. Here is the source code for the `ejbCreate` method in the `AccountEJB` class:

```
public String ejbCreate(String id, String firstName,
       String lastName, double balance)
       throws CreateException {

       if (balance < 0.00) {
          throw new CreateException
             ("A negative initial balance is not allowed.");
       }

       try {
          insertRow(id, firstName, lastName, balance);
       } catch (Exception ex) {
           throw new EJBException("ejbCreate: " +
              ex.getMessage());
       }

       this.id = id;
       this.firstName = firstName;
       this.lastName = lastName;
       this.balance = balance;

       return id;
    }
```

Although the `AccountEJB` class has just one `ejbCreate` method, an enterprise bean may contain multiple `ejbCreate` methods. For an example, see the `CareEJB.java` source code.

When writing an `ejbCreate` method for an entity bean, be sure to follow these rules:

- The access control modifier must be `public`.
- The return type must be the primary key (only for bean-managed persistence).
- The arguments must be legal types for Java RMI.
- The method modifier cannot be `final` or `static`.

The `throws` clause may include the `javax.ejb.CreateException` and other exceptions that are specific to your application. An `ejbCreate` method usually throws a `CreateException` if an input parameter is invalid. If an `ejbCreate` method cannnot create an entity because another entity with the same primary key already exists, it should throw a `javax.ejb.DuplicateKeyException` (a subclass of `CreateException`). If a client receives a `CreateException` or a `DuplicateKeyException`, it should assume that the entity was not created.

The state of an entity bean may be directly inserted into the database by a non-J2EE application. For example, a SQL script might insert a row into the `ACCOUNT` table. Although the entity bean for this row was not created by an `ejbCreate` method, the bean can be located by a client program.

The ejbPostCreate Method

For each `ejbCreate` method, you must write an `ejbPostCreate` method in the entity bean class. The EJB container invokes `ejbPostCreate` immediately after it calls `ejbCreate`. Unlike the `ejbCreate` method, the `ejbPostCreate` method can invoke the `getPrimaryKey` and `getEJBObject` methods of the `EntityContext` interface. (For more information on the `getEJBObject` method, see the section, “Passing an Entity Bean’s Object Reference” on page 72.) Often, your `ejbPostCreate` methods will be empty.

The signature of an `ejbPostCreate` must meet the following requirements:

- The number and types of arguments must match a corresponding `ejbCreate` method.
- The access control modifier must be `public`.
- The method modifier cannot be `final` or `static`.
- The return type must be `void`.

The `throws` clause may include the `javax.ejb.CreateException`, and other exceptions that are specific to your application.

The ejbRemove Method

A client removes an entity bean by invoking the `remove` method. This invocation causes the EJB client to call the `ejbRemove` method, which deletes the entity state from the database. The code for the `ejbRemove` method in the `AccountEJB` class follows:

```
public void ejbRemove() {


      try {
         deleteRow(id);
       } catch (Exception ex) {
            throw new EJBException("ejbRemove: " +
               ex.getMessage());
       }
   }
```

If the `ejbRemove` method encounters a system problem, it should throw the `javax.ejb.EJBException`. If it encounters an application error, it should throw a `javax.ejb.RemoveException`. (For a comparison of system and application exceptions, see the section "Handling Exceptions" on page 63.)

An entity bean may also be removed directly by a database deletion. For example, if a SQL script deletes a row that contains an entity bean state, then that entity bean is removed.

The ejbLoad Method and ejbStore Methods

If the EJB container needs to synchronize the instance variables of an entity bean with the corresponding values stored in a database, it invokes the `ejbLoad` and `ejbStore` methods. The `ejbLoad` method refreshes the instance variables from the database, and the `ejbStore` method writes the variables to the database. The client may not call `ejbLoad` and `ejbStore`.

If a business method is associated with a transaction, the container invokes `ejbLoad` before the business method executes. Immediately after the business method executes, the container calls `ejbStore`. Because the container invokes `ejbLoad` and `ejbStore`, you do not have to refresh and store the instance variables in your business methods-- the container performs these functions for you. The `AccountEJB` class relies on the container to synchronize the instance variables with the database. Therefore, the business methods of `AccountEJB` should be associated with transactions. (For instructions on setting transaction attributes for methods, see the section, "Running the New Enterprise Bean Wizard" on page 57.)

If the ejbLoad and ejbStore methods cannot locate an entity in the underlying database, they should throw the javax.ejb.NoSuchEntityException. This exception is a subclass of EJBException. Because EJBException is a subclass of RuntimeException, you do not have to include it in the throws clause. When NoSuchEntityException is thrown, the EJB container wraps it in a RemoteException before returning it to the client.

In the AccountEJB class, ejbLoad invokes the loadRow method, which issues a SQL select statement and assigns the retrieved data to the instance variables. The ejbStore method calls the storeRow method, which stores the instance variables in the database with a SQL update statement. Here is the code for ejbLoad and ejbStore methods:

```
public void ejbLoad() {

      try {
         loadRow();
       } catch (Exception ex) {
            throw new EJBException("ejbLoad: " +
               ex.getMessage());
       }
   }

   public void ejbStore() {

      try {
         storeRow();
       } catch (Exception ex) {
            throw new EJBException("ejbLoad: " +
               ex.getMessage());
       }
   }
```

The Finder Methods

The finder methods allow clients to locate entity beans. The AccountClient program locates entity beans with three finder methods:

```
Account jones = home.findByPrimaryKey("836");
. . .
Collection c = home.findByLastName("Smith");
```

```
. . .
Collection c = home.findInRange(20.00, 99.00);
```

For every finder method available to a client, the entity bean class must implement a corresponding method that begins with the prefix `ejbFind`. The `AccountEJB` entity bean class, for example, implements the `ejbFindByLastName` method as follows:

```
public Collection ejbFindByLastName(String lastName)
   throws FinderException {

   Collection result;

   try {
      result = selectByLastName(lastName);
    } catch (Exception ex) {
        throw new EJBException("ejbFindByLastName " +
           ex.getMessage());
    }

   if (result.isEmpty()) {
      throw new ObjectNotFoundException("No rows found.");
   }
   else {
      return result;
   }
}
```

The finder methods specific to your application, such as `ejbFindByLastName` and `ejbFindInRange`, are optional, but the `ejbFindByPrimaryKey` method is required. As its name infers, the `ejbFindByPrimaryKey` method accepts as an argument the primary key, which it uses to locate an entity bean. In the `AccountEJB` class, the primary key is the `id` variable. Here is the code for the `ejbFindByPrimaryKey` method:

```
public String ejbFindByPrimaryKey(String primaryKey)
   throws FinderException {

   boolean result;

   try {
      result = selectByPrimaryKey(primaryKey);
```

```
    } catch (Exception ex) {
        throw new EJBException("ejbFindByPrimaryKey: " +
           ex.getMessage());
    }

   if (result) {
      return primaryKey;
   }
   else {
      throw new ObjectNotFoundException
         ("Row for id " + primaryKey + " not found.");
   }
}
```

The `ejbFindByPrimaryKey` method may look strange to you, because it uses a `primaryKey` for both the method argument and return value. However, remember that the client does not call `ejbFindByPrimaryKey` directly. It is the EJB container that calls the `ejbFindByPrimaryKey` method. The client invokes the `findByPrimaryKey` method, which is defined in the home interface.

The following list summarizes the rules for the finder methods that you implement in an entity bean class with bean-managed persistence:

- The `ejbFindByPrimaryKey` method must be implemented.
- A finder method name must start with the prefix `ejbFind`.
- The access control modifier must be `public`.
- The method modifier cannot be `final` or `static`.
- The arguments and return type must be legal types for Java RMI.
- The return type must be the primary key or a collection of primary keys.

The `throws` clause may include the `javax.ejb.FinderException`, and other exceptions that are specific to your application. If a finder method returns a single primary key, it should throw the `javax.ejb.ObjectNotFoundException` if the requested entity does not exist. The `ObjectNotFoundException` is a subclass of `FinderException`. If a finder method returns a collection of primary keys, it should throw a `FinderException`.

The Business Methods

The business methods contain the business logic that you want to encapsulate within the entity bean. Usually, the business methods do not access the database, allowing you to separate business logic from the database access code. The `AccountEJB` entity bean contains these business methods:

```
public void debit(double amount)
       throws InsufficientBalanceException {

       if (balance - amount < 0) {
           throw new InsufficientBalanceException();
       }
       balance -= amount;
    }

    public void credit(double amount) {

       balance += amount;
    }

    public String getFirstName() {

       return firstName;
    }

    public String getLastName() {

       return lastName;
    }

    public double getBalance() {

       return balance;
    }
```

The `AccountClient` program invokes the business methods as follows:

```
Account duke = home.create("123", "Duke", "Earl");
duke.credit(88.50);
```

```
duke.debit(20.25);
double balance = duke.getBalance();
```

The requirements for the signature of a business method are the same for both session and entity beans:

- The method name must not conflict with a method name defined by the EJB architecture. For example, you cannot call a business method `ejbCreate` or `ejbActivate`.
- The access control modifier must be `public`.
- The method modifier cannot be `final` or `static`.
- The arguments and return types must be legal types for Java RMI.

The `throws` clause may include the the exceptions that you define for your application. The `debit` method, for example, throws the `InsufficientBalanceException`. To indicate a system-level problem, a business method should throw the `javax.ejb.EJBException`.

Database Calls

The following table summarizes the database access calls in the `AccountEJB` class:

TABLE 4-1 SQL Statement in `AccountEJB`

Method	Resulting SQL Statement
ejbCreate	insert
ejbFindByPrimaryKey	select
ejbFindByLastName	select
ejbFindInRange	select
ejbLoad	select
ejbRemove	delete
ejbStore	update

The business methods of the `AccountEJB` class are absent from the preceding table because they do not access the database. Instead, these business methods update the instance variables, which are written to the database when the EJB container calls

ejbStore. Another developer may have chosen to access the database in the business methods of the AccountEJB class. It's a design decision that depends on the specific needs of your application.

Before accessing a database you must connect to it. See the Database Connections chapter for instructions.

Home Interface

The home interface defines the methods that allow a client to create and find an entity bean. The AccountHome interface follows:

```
import java.util.Collection;
import java.rmi.RemoteException;
import javax.ejb.*;

public interface AccountHome extends EJBHome {

    public Account create(String id, String firstName,
                          String lastName)
        throws RemoteException, CreateException;

    public Account findByPrimaryKey(String id)
        throws FinderException, RemoteException;

    public Collection findByLastName(String lastName)
        throws FinderException, RemoteException;

    public Collection findInRange(double low, double high)
        throws FinderException, RemoteException;
}
```

The create methods in the home interface must conform to these requirements:

- It has the same number and types of arguments as its matching ejbCreate method in the enterprise bean class.
- It returns the remote interface type of the enterprise bean.
- The throws clause includes the exceptions specified by the throws clause of the corresponding ejbCreate and ejbPostCreate methods.

The `throws` clause contains the `java.rmi.RemoteException` and the `javax.ejb.CreateException`.

Every finder method in the home interface corresponds to a finder method in the entity bean class. The name of a finder method in the home interface begins with `find`, whereas the name of one in the entity bean class begins with `ejbFind`. For example, the `AccountHome` class defines the `findByLastName` method, and the `AccountEJB` class implements the `ejbFindByLastName` method. The rules for defining the signatures of the finder methods of a home interface follow:

- The number and types of arguments must match those of the corresponding method in the entity bean class.
- The return type is the entity bean's remote interface type, or a collection of those types.
- The exceptions in the `throws` clause include those of the corresponding method in the entity bean class.
- The `throws` clause contains the `javax.ejb.FinderException` and the `javax.ejb.RemoteException`.

Remote Interface

The remote interface extends `javax.ejb.EJBObject` and defines the business methods that a client may invoke. Here is the `Account` remote interface:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Account extends EJBObject {

    public void debit(double amount)
        throws InsufficientBalanceException, RemoteException;

    public void credit(double amount)
        throws RemoteException;

    public String getFirstName()
        throws RemoteException;

    public String getLastName()
        throws RemoteException;
```

```
    public double getBalance()
        throws RemoteException;
}
```

The requirements for the method definitions in a remote interface are the same for both session and entity beans:

- Each method in the remote interface must match a method in the enterprise bean class.
- The signatures of the methods in the remote interface must be identical to the signatures of the corresponding methods in the enterprise bean class.
- The arguments and return values must be valid RMI types.
- The `throws` clause must include `java.rmi.RemoteException`

Tips on Running the AccountEJB Example

Setting Up the Database

The instructions that follow explain how to use the `AccountEJB` example with a Cloudscape database. The Cloudscape software is included with the J2EE SDK download bundle. You may also run this example with databases provided by other vendors.

1. From the command-line prompt, run the Cloudscape database server:

```
cloudscape -start
```

(For more information, see the section, “Cloudscape Server” on page 166.)

2. Edit the script that creates the account database table.

UNIX:

```
cd $J2EE_HOME/doc/guides/ejb/examples/util
```

In the `cloudTable.sh` script, change <installation-location> to the directory in which you installed the J2EE SDK.

Windows:

```
cd %J2EE_HOME%\doc\guides\ejb\examples\util
```

In the `cloudTable` script, change <installation-location> to the directory in which you installed the J2EE SDK.

3. Run the script that creates the account database table.

UNIX:

```
cd $J2EE_HOME/doc/guides/ejb/examples/account
../util/cloudTable.sh
```

Windows:

```
cd %J2EE_HOME%\doc\guides\ejb\examples\account
..\util\cloudTable
```

Note: If you are not using a Cloudscape database, you may run the `account/createTable.sql` script to create the account table.

Running the New Enterprise Bean Wizard

The material in this section highlights the wizard steps that are unique to an entity bean with bean-managed persistence, such as `AccountEJB`. (For an introduction to the wizard, see the Getting Started chapter.)

General Dialog Box:

a. Select the Entity radio button.

b. In the Display Name field, enter `AccountBean`.

Entity Settings Dialog Box:

Select the radio button labelled “Bean managed persistence.”

Resource References Dialog Box:

a. Click Add.

b. In the Coded Name field, enter `jdbc/AccountDB`.

c. In the Type column, select javax.sql.DataSource.

d. In the Authentication column, select Container.

Transaction Management Dialog Box:

For the business methods, in the Transaction Type column select Required. (The business methods are `debit`, `credit`, `getFirstName`, `getLastName`, and `getBalance`.)

Deploying the J2EE Application

1. Click the radio button labelled “Return Client Jar.”

2. In the second dialog box, for the AccountBean entry in the Component / Reference Name field, enter `MyAccount` in the JNDI Name field.

3. For the jdbc / AccountDB entry, enter `jdbc/Cloudscape` in the JNDI Name field.

A Container-Managed Persistence Example

The entity bean discussed in this section represents a product. The source code files for this entity bean reside in the `doc/guides/ejb/examples/product` directory:

- Entity Bean Class (`ProductEJB.java`)
- Home Interface (`ProductHome.java`)
- Remote Interface (`Product.java`)
- A client class (`ProductClient.java`)

The code in the home and remote interfaces is the same whether or not an entity bean uses container-managed persistence. However, the code in the entity bean class is different for container-managed and bean-managed persistence. With container-managed persistence, the entity bean class contains no database access code. The Application Deployment Tool generates the SQL statements needed by the entity bean class. In order to generate the SQL statements, the tool needs to know which instance variables must be stored in the database. These instance variables are called container-managed fields.

Container-Managed Fields

The `ProductEJB` class has the following container-managed fields:

```
public String productId;
public String description;
public double price;
```

These fields represent the state of an `ProductEJB` instance. You specify the container-managed fields with the Application Deployment Tool (either in the New Enterprise Bean Wizard or on the Entity tab). A container-managed field must be one of the following types:

- Java serializable class
- Java primitive
- Reference of a home interface

- Reference of a remote interface

A container-managed field must be `public` and may not be defined as `transient`.

Using the Application Deployment tool, you define one (or more) of the container-managed fields as the primary key field (or fields) of the entity bean. For more information, see the section, “Primary Key Class” on page 66.

Entity Bean Class

The `ProductEJB` class relies on container-managed persistence. Because it contains no database access routines, the ProductEJB class is quite brief.

The ejbCreate Method

The `ejbCreate` method initializes the container-managed fields from the input parameters. The method returns null because with container-managed persistence the container ignores its return value. After the `ejbCreate` method executes, the container inserts the container-manged fields into the database. Here is the `ejbCreate` method of the `ProductEJB` class:

```
public String ejbCreate(String productId, String description,
      double price) throws CreateException {

      if (productId == null) {
         throw new CreateException("The productId is required.");
      }

      this.productId = productId;
      this.description = description;
      this.price = price;

      return null;
}
```

The ejbRemove Method

When the client invokes the `remove` method, the container calls the `ejbRemove` method. After the `ejbRemove` method returns, the container deletes the row from the database. If the container fails to delete the row, it throws an exception.

If an entity bean needs to perform some operation immediately before removal, it should do so in the `ejbRemove` method. Because the `ProductEJB` class does not have to perform such an operation, its `ejbRemove` method is empty.

The ejbLoad Method

When the container needs to refresh the entity bean's state from the database, it performs these steps:

- Selects the row from the database
- Assigns the row's column values to the container-manged fields
- Invokes the `ejbLoad` method

Usually, the `ejbLoad` method is empty. The entity bean may use the `ejbLoad` method, however, to transform the values read from the database. For example, the `ejbLoad` method might uncompress text data so that it can be manipulated by the business methods.

The ejbStore Method

When the container needs to save the entity bean's state in the database, it performs these steps:

- Invokes the `ejbStore` method
- Gets the values of the container-managed fields
- Updates the row in the database with the values of the container-managed fields

Like the `ejbLoad` method, the `ejbStore` method is typically empty. But if you need to transform container-managed fields before the container stores them in the database, you should do so in the `ejbStore` method. For example, the `ejbStore` method might compress text data before the container stores it in the database.

The Finder Methods

The `ProductHome` interface defines the following finder methods:

```
public Product findByPrimaryKey(String productId)
   throws FinderException, RemoteException;

public Collection findByDescription(String description)
   throws FinderException, RemoteException;
```

```
public Collection findInRange(double low, double high)
   throws FinderException, RemoteException;
```

Because the ProductEJB class uses container-managed persistence, it does not implement these finder methods. The Application Deployment tool implements the findByPrimaryKey method, including the SQL select statement that retrieves the row from the database. The tool also implements the customized finder methods (findByDescription and findInRange), but you must specify the where clauses for their select statements. For instructions on specifying the where clause, see the section, “Specifying the Deployment Settings” on page 62.

Table Creation

The Application Deployment Tool generates the SQL create table statement for an entity bean with container-managed persistence. For example, the tool generates the following statement for the ProductEJB class:

```
CREATE TABLE "ProductEJBTable"  (
"description" VARCHAR(255) ,
"price" DOUBLE PRECISION NOT NULL ,
"productId" VARCHAR(255),
CONSTRAINT "pk_ProductEJBTable" PRIMARY KEY ("productId") )
```

Note: The double quotes are part of the table, column, and constraint names. In the preceding SQL statement, the table name is “ProductEJBTable”-- not ProductEJBTable.

By default, the EJB container executes the create table statement when you deploy the application containing the entity bean. (Also by default, it drops the table when you uninstall the application.) If you do not want the container to create the table during deployment, follow these steps:

1. In the Application Deployment Tool, select the Entity tab of the entity bean.

2. In the Entity tabbed pane, click Deployment Settings.

3. In the Deployment Settings dialog box, de-select the checkbox for “Create Table on Deploy.”

Tips on Running the ProductEJB Example

Setting Up the Database

1. At the command-line prompt, run the Cloudscape database server:

```
cloudscape -start
```

2. Do not create the database table. The EJB container will create the table automatically.

Running the New Enterprise Bean Wizard

The material in this section highlights the wizard steps that are unique to an entity bean with container-managed persistence, such as `ProductEJB`.

General Dialog Box:

a. Select the Entity radio button.

b. In the Display Name field, enter `ProductBean`.

Entity Settings Dialog Box:

a. Select the radio button labelled "Container-Managed Persistence."

b. Select the check boxes for these container-managed fields: `productId`, `description`, and `price`.

c. In the Primary Key Class field, enter `java.lang.String`.

d. In the Primary Key Field Name field, select productId.

Transaction Management Dialog Box:

For the business methods, in the Transaction Type column select Required. (The business methods are `getDescription`, `getPrice`, and `setPrice`.)

Specifying the Deployment Settings

1. In the Application Deployment Tool, select the Entity tabbed pane for the ProductBean.

2. In the Entity tabbed pane, click Deployment Settings.

3. In the Database JNDI Name field, enter `jdbc/Cloudscape`.

4. Verify that the checkbox is selected for “Create Table on Deploy.”

5. Click Generate SQL Now.

6. A window pops open to inform you that the SQL statements have been generated for the Cloudscape database. Click OK.

7. A window pops open to inform you that you need to provide the SQL `where` clause for the `findByDescription` and `findInRange` methods. Click OK.

8. In the EJB method list, select `findByDescription`. A partial `select` statement appears.

9. Add the following `where` clause to the `select` statement:

```
WHERE “description” = ?1
```

Note: You must include the quotes in the `“description”` column name. The `?1` represents the first parameter of the finder method.

10. In the EJB method list, select `findInRange`. A partial select statement appears.

11. Add the following `where` clause to the `select` statement:

```
WHERE “price” BETWEEN ?1 AND ?2
```

Note: You must include the quotes in the `“price”` column name. The `?1` and `?2` represent the first and second parameters of the finder method.

12. Click OK.

Deploying the J2EE Application

1. Click the radio button labelled “Return Client Jar.”

2. In the second dialog box, for the ProductBean enter `MyProduct` in the JNDI Name field.

Handling Exceptions

The exceptions thrown by enterprise beans fall into two categories: system and application.

A system exception indicates a problem with the services that support an application. Examples of these problems include the following: a database connection cannot be obtained, a SQL `insert` fails because the database is full, a

`lookup` method cannot find the desired object. If your enterprise bean encounters a sytem-level problem, it should throw a `javax.ejb.EJBException`. The container will wrap the `EJBException` in a `RemoteException`, which it passes back to the client. Because the `EJBException` is a subclass of the `RuntimeException`, you do not have to specify it in the throws clause of the method declaration. If a system exception is thrown, the EJB container might destroy the bean instance. Therefore, a system exception cannot be handled by the bean's client program; it requires intervention by a system administrator.

An application exception signals an error in the business logic of an enterprise bean. There are two types of application exceptions: customized and predefined. A customized exception is one that you've coded yourself, such as the `InsufficentBalanceException` thrown by the `debit` business method of the `AccountEJB` example. The `javax.ejb` package includes several predefined exceptions that are designed to handle common problems. For example, an `ejbCreate` method should throw a `CreateException` to indicate an invalid input parameter. When an enterprise bean throws an application exception, the container does not wrap it in another exception. The client should be able to handle any application exception it receives.

If a system exception occurs within a transaction, the EJB container rolls back the transaction. However, if an application exception is thrown within a transaction, the container does not roll back the transaction.

The following table summarizes the exceptions of the `javax.ejb` package. All of these exceptions are application exceptions, except for the `NoSuchEntityException` and the `EJBException`, which are system exceptions.

TABLE 4-2 Exceptions

Method Name	Exception It Throws	Reason for Throwing
ejbCreate	CreateException	An input parameter is invalid.
ejbFindByPrimaryKey (and other finder methods)	ObjectNotFoundException (subclass of FinderException)	The database row for the requested entity bean is cannot be found.
ejbRemove	RemoveException	The entity bean's row cannot be deleted from the database.

TABLE 4-2 Exceptions

Method Name	Exception It Throws	Reason for Throwing
ejbLoad	NoSuchEntityException	The database row to be loaded cannot be found.
ejbStore	NoSuchEntityException	The database row to be updated cannot be found.
(all methods)	EJBException	A system problem has been encountered.

Primary Key Class

You specify the primary key class with the Application Deployment Tool. When deploying the `ProductEJB` bean, for example, you would specify a `java.lang.String` as the primary key class. In most cases, your primary key class will be a `String` or some other class that belongs to the `java` package.

Creating a Primary Key Class

For some entity beans, you will need to define your own primary key class. For example, if a primary key is composed of multiple fields then you must create a primary key class. In the following primary key class, the `productId` and `vendorId` fields together uniquely identify an entity bean:

```
public class ItemKey implements java.io.Serializable {

   public String productId;
   public String vendorId;

   public ItemKey() { };

   public ItemKey(String productId, String vendorId) {

     this.productId = productId;
     this.vendorId = vendorId;
   }

   public String getProductId() {

      return productId;
   }

   public String getVendorId() {

      return vendorId;
   }
```

```
   public boolean equals(Object other) {

      if (other instanceof ItemKey) {
         return (productId.equals(((ItemKey)other).productId)
                 && vendorId.equals(((ItemKey)other).vendorId));
      }

      return false;
   }

   public int hashCode() {

      return productId.hashCode();
   }
}
```

Class Requirements

A primary key class must meet these requirements:

- The access control modifier of the class is `public`.
- All fields are declared as `public`.
- For container-managed persistence, the field names in the primary key clsss must match the corresponding container-managed fields in the entity bean class.
- The class has a `public` default constructor.
- The class implements the `hashCode()` and `equals(Object other)` methods.
- The class is serializable.

Bean-Managed Persistence and the Primary Key Class

With bean-managed persistence, the `ejbCreate` method returns the primary key class:

```
public ItemKey ejbCreate(String productId, String vendorId,
   String description) throws CreateException {

   if (productId == null || vendorId == null) {
```

```
      throw new CreateException(
                "The productId and vendorId are required.");
   }

   this.productId = productId;
   this.vendorId = vendorId;
   this.description = description;

   return new ItemKey(productId, vendorId);
}
```

The `ejbFindByPrimaryKey` verifies the existence of the database row for the given primary key:

```
public ItemKey ejbFindByPrimaryKey(ItemKey primaryKey)
   throws FinderException {

   try {
      if (selectByPrimaryKey(primaryKey))
         return primaryKey;
   . . .
}

private boolean selectByPrimaryKey(ItemKey primaryKey)
   throws SQLException {

   String selectStatement =
         “select productid “ +
         “from item where productid = ? and vendorid = ?”;
   PreparedStatement prepStmt =
         con.prepareStatement(selectStatement);
   prepStmt.setString(1, primaryKey.getProductId());
   prepStmt.setString(2, primaryKey.getVendorId());
   ResultSet rs = prepStmt.executeQuery();
   boolean result = rs.next();
   prepStmt.close();
   return result;
}
```

Container-Managed Persistence and the Primary Key Class

For an entity bean with container-managed persistence, the `ejbCreate` method returns null. (With bean-managed persistence, it returns an instance of the primary key class.)

With container-managed persistence, you do not have to write the code for the `ejbFindByPrimaryKey` method. (For more information, see the section, “The Finder Methods” on page 60.)

To create an container-managed entity bean with the `ItemKey` class example, in the Entity Settings dialog of the New Enterprise Bean Wizard you would specify these settings:

- Select Container-Managed Persistence.
- Select the check boxes for all three instance variables (`productId`, `vendorId`, `description`).
- In the PrimaryKey Class field, enter `ItemKey`.
- Leave the Primary Key Field Name field blank.

Getting the Primary Key

A client can fetch the primary key of an entity bean by invoking the `getPrimaryKey` method of the `EJBObject` class:

```
Account account;
. . .
String id = (String)account.getPrimaryKey();
```

The entity bean retrieves its own primary key by calling the `getPrimaryKey` method of the `EntityContext` class:

```
EntityContext context;
. . .
String id = (String) context.getPrimaryKey();
```

The Life Cycle of an Entity Bean

The life cycle of an entity bean is controlled by the EJB container, not by your application. However, you may find it helpful to learn about the life cycle when deciding in which method your entity bean will connect to a database. (See the Database Connections chapter for more information.)

Figure 4-1 shows the stages that an entity bean passes through during its lifetime. After the EJB container creates the instance, it calls the `setEntityContext` method of the entity bean class. The `setEntityContext` method passes the entity context to the bean.

After instantiation, the entity bean moves to a pool of available instances. While in the pooled stage, the instance is not associated with any particular EJB object identity. All instances in the pool are identical. The EJB container assigns an identity to an instance when moving it to the ready stage.

There are two paths from the pooled stage to the ready stage. On the first path, the client invokes the `create` method, causing the EJB container to call the `ejbCreate` and `ejbPostCreate` methods. On the second path, the EJB container invokes the `ejbActivate` method. While in the ready stage, an entity bean's business methods may be invoked.

There are also two paths from the ready stage to the pooled stage. First, a client may invoke the `remove` method, which causes the EJB container to call the `ejbRemove` method. Second, the EJB container may invoke the `ejbPassivate` method.

At the end of the life cycle, the EJB container removes the instance from the pool and invokes the `unsetEntityContext` method.

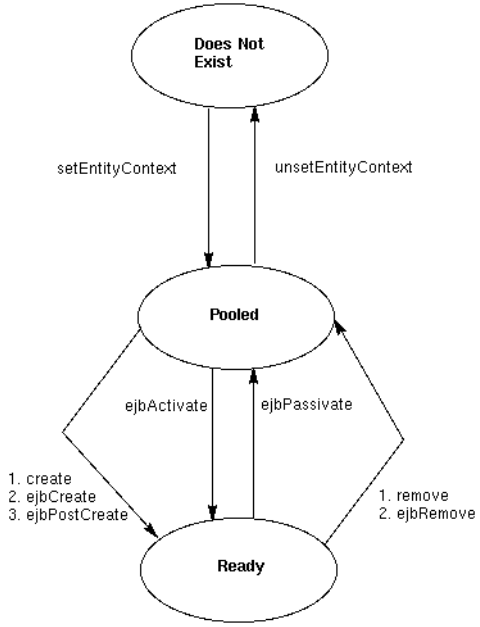

FIGURE 4-1 Life Cycle of an Entity Bean

In the pooled state, an instance is not associated with any particular EJB object identity. With bean-managed persistence, when the EJB container moves an instance from the pooled state to the ready state, it does not automatically set the primary key. Therefore, the `ejbCreate` and `ejbActivate` methods must set the primary key. If the primary key is incorrect, the `ejbLoad` and `ejbStore` methods cannot synchronize the instance variables with the database. In the `AccountEJB` example, the `ejbCreate` method assigns the primary key from one of the input parameters. The `ejbActivate` method sets the primary key (`id`) as follows:

```
id = (String)context.getPrimaryKey();
```

In the pooled state, the values of the instance variables are not needed. You can make these instance variables eligible for garbage collection by setting them to null in the `ejbPasssivate` method.

Comparing Entity Beans

A client can determine if the object references of two entity beans are identical by invoking the `isIdentical` method:

```
Account accta, acctb;
. . .
if (accta.isIdentical(acctb))
   System.out.println("identical");
```

Or, the client can fetch and compare the primary keys of two entity beans:

```
String key1 = (String)accta.getPrimaryKey();
String key2 = (String)acctb.getPrimaryKey();

if (key1.compareTo(key2) == 0)
   System.out.println("equal");
```

Passing an Entity Bean's Object Reference

If you want to pass an entity bean to another bean, you can't pass the `this` reference. Instead, you must pass the entity bean's object reference. The following code snippet shows how to retrieve the object reference by calling `getEJBObject` on the `EntityContext` of an entity bean.

```
public void setEntityContext(EntityContext ec) {
   // save the entity context in an instance variable
   this.context = ec;
}
   . . .
public void passItOn(Inventory tally) {
   // pass the object reference
    tally.copyItems(context.getEJBObject());
}
```

CHAPTER 5

Database Connections

The EJB container maintains the pool of database connections. This pool is transparent to the enterprise beans. When an enterprise bean requests a connection, the container fetches one from the pool and assigns it to the bean. Because the time-consuming connection has already been made, the bean quickly gets a connection. The bean may release the connection after each database call, since it can rapidly get another connection. And because such a bean holds the connection for a short time, the same connection may be shared sequentially by many beans.

The persistence type of the enterprise bean determines whether or not you code the connection routine. You must code the connection for enterprise beans that access a database and do not have container-managed persistence. Such beans include entity beans with bean-managed persistence and session beans. For entity beans with container-managed persistence, the Application Deployment Tool generates the connect routines for you.

Coded Connections

The bean routine that connects to the database should not hardcode the actual name (URL) of the database. Instead, it should refer to the database with a logical name and use a JNDI `lookup` when obtaining the database connection. This level of indirection provides several benefits:

- You can deploy the same enterprise bean in different environments that have databases with different names.
- You can re-use the enterprise bean in multiple applications.
- You can assemble the enterprise beans into applications that run in a distributed environment. (The enterprise beans and the databases they access may run on different machines.)

The instructions that follow show you how to link the logical database name in your code with the JNDI name of the resource manager connection factory-- a term that may be new to you. A *resource manager* is a storage mechanism such as a DBMS. A *resource manager connection* is an object, such as `java.sql.Connection`, that represents a session with the resource manager (DBMS). A *resource manager connection factory* is an object that creates resource manager connections. For example, a `javax.sql.DataSource` object is a connection factory because it creates a `java.sql.Connection` object.

How to Connect

The code examples in this section are from the the `AccountEJB` class, which was described in the the Entity Beans chapter. Because the `AccountEJB` class uses bean-managed persistence, it connects to the database with the following steps:

1. Specify the logical database name.

```
private String dbName = "java:comp/env/jdbc/AccountDB";
```

The `java:comp/env/` prefix is the JNDI context for the component. The `jdbc/AccountDB` string is the logical database name.

2. Obtain the `DataSource` associated with the logical name.

```
InitialContext ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup(dbName);
```

3. Get the `Connection` from the `DataSource`.

```
Connection con =  ds.getConnection();
```

When To Connect

When coding an enterprise bean, you must decide how long the bean will retain the connection. Generally you have two choices: either hold the connection for the lifetime of the bean, or only during each database call. Your choice determines the method (or methods) in which your bean connects to a database.

Longterm Connections

You can design an enterprise bean that holds a database connection for its entire lifetime. Because the bean connects and disconnects just once, its code is slightly easier to write. But there's a tradeoff-- other enterprise beans may not acquire the connection. Session and entity beans issue the lifelong connections in different methods.

Session Beans

The EJB container invokes the `ejbCreate` method at the beginning of a session bean's life cycle, and invokes the `ejbRemove` method at the end. To retain a connection for the lifetime of a session bean, you connect to the database in `ejbCreate` and disconnect in `ejbRemove`. If the session bean is stateful, you must also connect in `ejbActivate` and disconnect in `ejbPassivate`. A stateful session bean requires these additional calls because the EJB container may passivate the bean during its lifetime. During passivation, a stateful session bean is saved in secondary storage, but a database connection may not be saved in this manner. Because a stateless session bean cannot be passivated, it does not require the additional calls in `ejbActivate` and `ejbPassivate`. For more information on activation and passivation, see the section, "The Life Cycle of a Session Bean" on page 38. For an example of a stateful session bean with a longterm connection, see the `TellerEJB.java` code.

Entity Beans

After instantiating an entity bean and moving it to the pooled stage, the EJB container invokes the `setEntityContext` method. Conversely, the EJB container invokes the `unsetEntityContext` method when the entity bean leaves the pooled stage and becomes eligible for garbage collection. To retain a database connection for its entire lifespan, an entity bean connects in the `setEntityContext` method and disconnects in the `unsetEntityContext` method. To see a diagram of the life cycle, refer to the section, "The Life Cycle of an Entity Bean" on page 70. For an example of an entity bean with a longterm connection, see the `AccountEJB.java` code.

Shortterm Connections

Briefly held connections allow many enterprise beans to share the same connection. Because the EJB container manages a pool of database connections, enterprise beans can quickly obtain and release the connections. For example, a business method might connect to a database, insert a row, and then disconnect.

In a session bean, a business method that connects to a database should be transactional. The transaction will help maintain data integrity.

Specifying the JNDI Name for Deployment

Specifying the JNDI name of the database is a two-step process:

1. Enter the coded name.

To enter the coded name in the bean's deployment descriptor, you run the Application Deployment Tool and step through the New Enterprise Bean Wizard. In the Coded Name field of the wizard's Resource References dialog box, you enter the database name from the bean's code. For example, the logical database name coded in the `AccountEJB` class is the `jdbc/AccountDB` portion of the following string:

```
private String dbName = "java:comp/env/jdbc/AccountDB";
```

In the Coded Name field of the wizard, you enter `jdbc/AccountDB`. In the Type column, select javax.sql.DataSource.

2. Map the coded name to the JNDI name.

The Application Deployment Tool allows you to map this information in two ways: in a deployment dialog box and in the JNDI Names tabbed pane of the J2EE application. If the `AccountEJB` class connects to a Cloudscape database, for example, you enter `jdbc/Cloudscape` in the JNDI Name field of the JNDI Names tabbed pane.

The J2EE server automatically enters the database JNDI names such as `jdbc/Cloudscape` into the name space. The server obtains these JNDI names from the `jdbc.datasources` entry of the `config/default.properties` file. The `jdbc.datasources` entry maps the JNDI name to the URL of the database. (For more information on `jdbc.datasources`, see the Configuration Guide.)

Specifying Database Users and Passwords

To connect to the Cloudscape database bundled with this release, you do not specify a database user and password. Authentication is performed by a separate service. (For more information about authentication, see the Security chapter.)

However, some types of databases do require a user and password during connection. For these databases, if the `getConnection` call has no parameters, you must specify the database user and password with the Application Deployment Tool. To specify these values, click on the Resource Ref's tabbed pane of the enterprise bean, select the appropriate row in the table labelled, "Resource Factories Referenced in Code," and enter the database user name and password in the fields at the bottom.

If you wish to obtain the database user and password programmatically, you do not need to specify them with the Application Deployment Tool. In this case, you include the database user and password in the arguments of the `getConnection` method:

```
con = dataSource.getConnection(dbUser, dbPassword);
```

Container-Managed Connections

With container-managed persistence, the entity bean class does not contain the code that connects to a database. Instead, this code is generated by the Application Deployment Tool. Using the tool, you specify the JNDI name of the database in the JNDI Name field of the Deployment Settings dialog box. To access this dialog box, in the Entity tabbed pane of the enterprise bean, click the Deployment Settings button.

The `ProductEJB` class, for instance, is an entity bean with container-managed persistence. To enable this bean to connect to the example Cloudscape database, you enter `jdbc/Cloudscape` in the JNDI Name field of the Deployment Settings dialog box.

CHAPTER 6

Transactions

To emulate a business transaction, a program may need to perform several steps. A financial program, for example, might transfer funds from a checking account to a savings account with the steps listed in the following pseudo-code.

```
begin transaction
   debit checking account
   credit savings account
   update history log
commit transaction
```

Either all three of these steps must complete, or none of them at all. Otherwise, data integrity is lost. Because the steps within a transaction are a unified whole, a *transaction* is often defined as an indivisible unit of work.

A transaction can end in two ways: with a commit or a rollback. When a transaction commits, the data modifications made by its statements are saved. If a statement within a transaction fails, the transaction rolls back, undoing the effects of all statements in the transaction. In the pseudo-code, for example, if a disk drive crashed during the credit step, the transaction rolls back and undoes the data modifications made by the debit statement. Although the transaction failed, data integrity is intact because the accounts still balance.

In the preceding pseudo-code, the begin and commit statments mark the boundaries of the transaction. When deploying an enterprise bean, you determine how the boundaries are set by specifying either container-managed or bean-managed transactions.

Container-Managed Transactions

In an enterprise bean with container-managed transactions, the EJB container sets the boundaries of the transactions. You can use container-managed transactions with both session and entity beans. Container-managed transactions simplify development because the enterprise bean code does not explicitly mark the transaction's boundaries. The code does not include statements that begin and end the transaction.

Typically, the container begins a transaction immediately before an enterprise bean method starts. It commits the transaction just before the method exits. Each method can be associated with a single transaction. Nested or multiple transactions are not allowed within a method.

Container-managed transactions do not require all methods to be associated with transactions. When deploying a bean, you specify which of the bean's methods are associated with transactions by setting the transaction attributes.

Transaction Attributes

A transaction attribute controls the scope of a transaction. Figure 6-1 illustrates why controlling the scope is important. In the diagram, method-A begins a transaction and then invokes method-B of Bean-2. When method-B executes, does it run within the scope of the transaction started by method-A or does it execute with a new transaction? The answer depends on the transaction attribute of method-B.

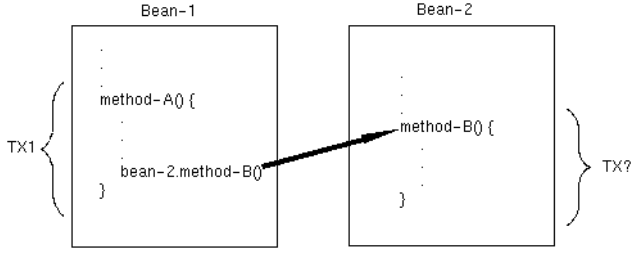

FIGURE 6-1 Transaction Scope

Transaction Attribute Values

A transaction attribute may have one of the following values:

- Required
- RequiresNew
- Mandatory
- NotSupported
- Supports
- Never

Required

If the client is running within a transaction and it invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.

The Required attribute will work for most transactions. Therefore, you may want to use it as a default, at least in the early phases of development. Because transaction attributes are declarative, you can easily change them at a later time.

RequiresNew

If the client is running within a transaction and it invokes the enterprise bean's method, the container takes the following steps:

- suspends the client's transaction
- starts a new transaction
- delegates the call to the method
- resumes the client's transaction after the method completes

If the client is not associated with a transaction, the container starts a new transaction before running the method.

You should use the RequiresNew attribute when you want to ensure that the method always runs within a new transaction.

Mandatory

If the client is running within a transaction and it invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container throws the `TransactionRequiredException`.

Use the Mandatory attribute if the enterprise bean's method must use the transaction of the client.

NotSupported

If the client is running within a transaction and it invokes the enterprise bean's method, the container suspends the client's transaction before invoking the method. After the method has completed, the container resumes the client's transaction.

If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Use the NotSupported attribute when you want to ensure that the method will never run within a transaction generated by the container.

Supports

If the client is running within a transaction and it invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Because the transactional behavior of the method may vary, you should use the Supports attribute with caution.

Never

If the client is running within a transaction and it invokes the enterprise bean's method, the container throws a `RemoteException`. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Summary of Transaction Attributes

Table 6-1 summarizes the effects of the transaction attributes. Both the T1 and T2 transactions are controlled by the container. A T1 transaction is associated with the client that calls a method in the enterprise bean. In most cases, the client is another enterprise bean. A T2 transaction is started by the container just before the method executes.

In the last column, the word "none" means that the business method does not execute within a transaction controlled by the container. However, the database calls in such a business method might be controlled by the transaction manager of the DBMS.

TABLE 6-1 Transaction Attributes and Scope

Transaction Attribute	Client's Transaction	Business Method's Transaction
Required	none	T2
	T1	T1
RequiresNew	none	T2
	T1	T2
Mandatory	none	error
	T1	T1
NotSupported	none	none
	T1	none
Supports	none	none
	T1	T1
Never	none	none
	T1	error

Setting Transaction Attributes

Because transaction attributes are stored in the deployment descriptor, they can be changed during several phases of J2EE application development: enterprise bean creation, application assembly, and deployment. However, as an enterprise bean developer, it is your responsibility to specify the attributes when creating the bean. The attributes should be modified only by an application developer who is assembling components into larger applications. Do not expect the person who is deploying the J2EE application to specify the transaction attributes.

You can specify the transaction attributes for the entire enterprise bean or for individual methods. If you've specified one attribute for a method and another for the bean, the attribute for the method takes precedence. When specifying attributes for individual methods, the requirements for session and entity beans vary. Session beans need the attributes defined for business methods, but do not allow them for the `create` methods. Entity beans require transaction attributes for the business, `create`, `remove`, and finder methods.

Rolling Back a Container-Managed Transaction

There are two ways to roll back a container-managed transaction. First, if a system exception is thrown, the container will automatically roll back the transaction. Second, by invoking the `setRollbackOnly` method of the `EJBContext` interface, the bean method instructs the container to roll back the transaction. If the bean throws an application exception, the roll back is not automatic, but may be initiated by a call to `setRollbackOnly`. (See the section "Handling Exceptions" on page 63 for a description of system and application exceptions.)

The `transferToSaving` method of the `BankEJB` example illustrates the `setRollbackOnly` method. If a negative checking balance occurs, `transferToSaving` invokes `setRollBackOnly` and throws an application exception (`InsufficientBalanceException`). The `updateChecking` and `updateSaving` methods update database tables. If the updates fail, these methods throw a `SQLException` and the `transferToSaving` method throws an `EJBException`. Because the `EJBException` is a system exception, it causes the container to automatically roll back the transaction. Here is the code for the `transferToSaving` method:

```
public void transferToSaving(double amount) throws
   InsufficientBalanceException  {

   checkingBalance -= amount;
   savingBalance += amount;

   try {
      updateChecking(checkingBalance);
      if (checkingBalance < 0.00) {
         context.setRollbackOnly();
         throw new InsufficientBalanceException();
      }
      updateSaving(savingBalance);
   } catch (SQLException ex) {
       throw new EJBException
          ("Transaction failed due to SQLException: "
          + ex.getMessage());
   }
}
```

When the container rolls back a transaction, it always undoes the changes to data made by SQL calls within the transaction. However, only in entity beans will the container undo changes made to instance variables. (It does so by automatically invoking the entity bean's `ejbLoad` method, which loads the instance variables from the database.) When a rollback occurs, a session bean must explicitly reset any instance variables changed within the transaction. The easiest way to reset a session bean's instance variables is by implementing the `SessionSynchonization` interface.

Synchronizing a Session Bean's Instance Variables

The `SessionSynchonization` interface, which is optional, allows you to synchronize the instance variables with their corresponding values in the database. The container invokes the `SessionSynchonization` methods-- `afterBegin`, `beforeCompletion`, and `afterCompletion`-- at each of the main stages of a transaction.

The `afterBegin` method informs the instance that a new transaction has begun. The container invokes `afterBegin` immediately before it invokes the business method. The `afterBegin` method is a good place to load the instance variables from the database. The `BankEJB` class, for example, loads the `checkingBalance` and `savingBalance` variables in the `afterBegin` method:

```
public void afterBegin() {


   System.out.println("afterBegin()");
   try {
      checkingBalance = selectChecking();
      savingBalance = selectSaving();
   } catch (SQLException ex) {
       throw new EJBException("afterBegin Exception: " +
           ex.getMessage());
   }
}
```

The container invokes the `beforeCompletion` method after the business method has finished, but just before the transaction commits. The `beforeCompletion` method is the last opportunity for the session bean to roll back the transaction (by calling `setRollbackOnly`). If it hasn't already updated the database with the values of the instance variables, the session bean may do so in the `beforeCompletion` method.

The `afterCompletion` method indicates that the transaction has completed. It has a single boolean parameter, whose value is true if the transaction was committed and false if it was rolled back. If a rollback occurred, the session bean can refresh its instance variables from the database in the `afterCompletion` method:

```
public void afterCompletion(boolean committed) {

   System.out.println("afterCompletion: " + committed);
   if (committed == false) {
      try {
         checkingBalance = selectChecking();
         savingBalance = selectSaving();
      } catch (SQLException ex) {
          throw new EJBException("afterCompletion SQLException: " +
             ex.getMessage());
      }
   }
}
```

Methods Not Allowed in Container-Managed Transactions

You should not invoke any method that might interfere with the transaction boundaries set by the container. The list of prohibited methods follows:

- `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection`
- `getUserTransaction` method of `javax.ejb.EJBContext`
- any method of `javax.transaction.UserTransaction`

You may, however, use these methods to set boundaries in bean-managed transactions.

Bean-Managed Transactions

In a bean-managed transaction, the session bean code invokes methods that mark the boundaries of the transaction. An entity bean may not have bean-managed transactions; it must use container-managed transactions instead. Although beans with container-managed transactions require less coding, they have one limitation:

When a method is executing, it can be associated with either a single transaction or no transaction at all. If this limitation will make coding your session bean difficult, you should consider using bean-managed transactions.

The following pseudo-code illustrates the kind of fine-grained control you can obtain with bean-managed transactions. By checking various conditions, the pseudo-code decides whether to start and stop different transactions within the business method.

```
begin transaction
...
update table-a
...
if (condition-x)
   commit transaction
else if (condition-y)
   update table-b
   commit transaction
else
   rollback transaction
   begin transaction
   update table-c
   commit transaction
```

When coding a bean-managed transaction, you must decide whether to use JDBC or JTA transactions. The sections that follow discuss the techniques and merits of both approaches.

JDBC Transactions

A JDBC transaction is controlled by the transaction manager of the DBMS. You may want to use JDBC transactions when wrapping legacy code inside a session bean. To code a JDBC transaction, you invoke the `commit` and `rollback` methods of the `javax.sql.Connection` interface. The beginning of a transaction is implicit. A transaction begins with the first SQL statement that follows the most recent `commit`, `rollback`, or `connect` statement. (This rule is generally true, but may vary with DBMS vendor.)

The following code is from the `WarehouseEJB` example, a session bean that uses the `Connection` interface's methods to delimit bean-managed transactions. The `ship` method starts by invoking `setAutoCommit` on the `Connection` object `con`. This invocation tells the DBMS not to automatically commit every SQL statement.

Next, the `ship` method calls routines that update the `order_item` and `inventory` database tables. If the updates succeed, the transaction is committed. But if an exception is thrown, the transaction is rolled back.

```
public void ship (String productId, String orderId, int quantity) {

   try {
      con.setAutoCommit(false);
      updateOrderItem(productId, orderId);
      updateInventory(productId, quantity);
      con.commit();
   } catch (Exception ex) {
       try {
          con.rollback();
          throw new EJBException("Transaction failed: " +
             ex.getMessage());
       } catch (SQLException sqx) {
           throw new EJBException("Rollback failed: " +
              sqx.getMessage());
       }
   }
}
```

JTA Transactions

JTA is the abbreviation for the Java Transaction API. This API allows you to demarcate transactions in a manner that is independent of the transaction manager implementation. The J2EE SDK implements the transaction manager with the Java Transaction Service (JTS). But your code doesn't call the JTS methods directly. Instead, it invokes the JTA methods, which then call the lower-level JTS routines.

A JTA transaction is controlled by the J2EE transaction manager. You may want to use a JTA transaction because it can span updates to multiple databases from different vendors. A particular DBMS's transaction manager may not work with heterogenous databases. However, the J2EE transaction manager does have one limitation-- it does not support nested transactions. (It cannot start a transaction for an instance until the previous transaction has ended.)

To demarcate a JTA transaction, you invoke the `begin`, `commit`, and `rollback` methods of the `UserTransaction` interface. The following code, taken from the `TellerEJB` example program, demonstrates the `UserTransaction` methods. The `begin` and `commit` invocations delimit the updates to the database. If the updates fail, the code invokes the `rollback` method and throws an `EJBException`.

```
public void withdrawCash(double amount) {

   UserTransaction ut = context.getUserTransaction();

   try {
      ut.begin();
      updateChecking(amount);
      machineBalance -= amount;
      insertMachine(machineBalance);
      ut.commit();
   } catch (Exception ex) {
       try {
          ut.rollback();
       } catch (SystemException syex) {
           throw new EJBException
              ("Rollback failed: " + syex.getMessage());
       }
       throw new EJBException
          ("Transaction failed: " + ex.getMessage());
    }
}
```

Returning Without Committing

In a stateless session bean with bean-managed transactions, a business method must commit or roll back a transaction before returning. However, a stateful session bean does not have this restriction.

In a stateful session bean with a JTA transaction, the association between the bean instance and the transaction is retained across multiple client calls. Even if each business method called by the client opens and closes the database connection, the association is retained until the instance completes the transaction.

In a stateful session bean with a JDBC transaction, the JDBC connection retains the association between the bean instance and the transaction across multiple calls. If the connection is closed, the association is not retained.

Methods Not Allowed in Bean-Managed Transactions

Do not invoke the `getRollbackOnly` and `setRollbackOnly` methods of the `EJBContext` interface. These methods should be used only in container-managed transactions. For bean-managed transactions you invoke the the `getStatus` and `rollback` methods of the `UserTransaction` interface.

Summary of Transaction Options

The decision tree in figure 6-2 shows the different approaches to transaction management that you may take. Your first choice depends on whether the enterprise bean is an entity or a session bean. An entity bean must use container-managed transactions. With container-managed transactions, you specify the transaction attributes in the deployment descriptor and you roll back a transaction with the `setRollbackOnly` method of the `EJBContext` interface. A session bean may have either container-managed or bean-managed transactions. There are two types of bean-managed transactions: JDBC and JTA transactions. You delimit JDBC transactions with the `commit` and `rollback` methods of the `Connection` interface. To demarcate JTA transactions, you invoke the `begin`, `commit`, and `rollback` methods of the `UserTransaction` interface.

In a session bean with bean-managed transactions, it is possible to mix JDBC and JTA transactions. This practice is not recommended, however, because it could make your code difficult to debug and maintain.

If you're unsure about how to set up transactions in an enterprise bean, here's a tip: In the deployment descriptor specify container-managed transactions. Then, set the Required transaction attribute for the entire bean. This approach will work most of the time.

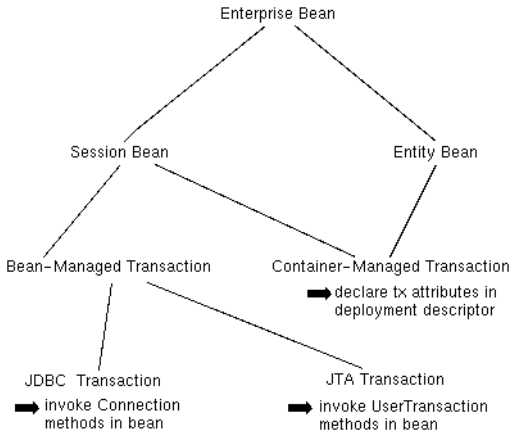

FIGURE 6-2 Options in Specifying Transactions

Transaction Timeouts

For container-managed transactions, you control the transaction timeout interval by setting the value of the `transaction.timeout` property in the `config/default.properties` file. For example, you would set the timeout value to 5 seconds as follows:

```
transaction.timeout=5
```

With this setting, if the transaction has not completed within 5 seconds, the EJB container manager rolls it back.

When J2EE is first installed, the timeout value is set to 0:

```
transaction.timeout=0
```

If the value is 0, the transaction will not time out.

Only enterprise beans with container-managed transactions are affected by the `transaction.timeout` property. For enterprise beans with bean-managed, JTA transactions, you invoke the `setTransactionTimeout` method of the `UserTransaction` interface.

Isolation Levels

Transactions not only ensure the full completion (or rollback) of the statements that they enclose, they also isolate the data modified by the statements. The isolation level describes the degree to which the data being updated is visible to other transactions.

Suppose that a transaction in one program updates a customer's phone number, but before the transaction commits another program reads the same phone number. Will the second program read the updated and uncommitted phone number or will it read the old one? The answer depends on the isolation level of the transaction. If the transaction allows other programs to read uncommitted data, performance may improve because the other programs don't have to wait until the transaction ends. But there's a tradeoff-- if the transaction rolls back, another program might read the wrong data.

You cannot modify the isolation level of a entity beans with container-managed persistence. These beans use the default isolation level of the DBMS, which is usually READ_COMMITTED.

For entity beans with bean-managed persistence and for all session beans, you can set the isolation level programmatically with the API provided by the underlying DBMS. A DBMS, for example, might allow you to permit uncommitted reads by invoking the `setTransactionIsolation` method:

```
Connection con;
...
con.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
```

Do not change the isolation level in the middle of a transaction. Usually, such a change causes the DBMS software to issue an implicit commit. Because the isolation levels offered by DBMS vendors may vary, you should check the DBMS documentation for more information.

Updating Multiple Databases

The J2EE transaction manager controls all enterprise bean transactions except for bean-managed JDBC transactions. The J2EE transaction manager allows an enterprise bean to update multiple databases within a transaction. The figures that follow show two scenarios for updating multiple databases in a single transaction.

In figure 6-3, the client invokes a business method in Bean-A. The business method begins a transaction, updates Database-X, updates Database-Y, and invokes a business method in Bean-B. The second business method updates Database-Z and returns control to the the business method in Bean-A, which commits the transaction. All three database updates occur in the same transaction.

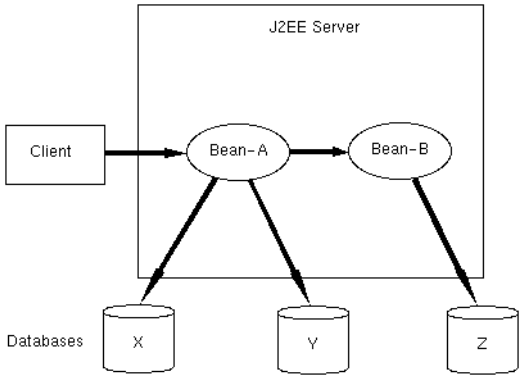

FIGURE 6-3 Updating Multiple Databases

In figure 6-4, the client calls a business method in Bean-A, which begins a transaction and updates Database-X. Then, Bean-A invokes a method in Bean-B, which resides in a remote J2EE server. The method in Bean-B updates Database-Y. The transaction managers of the J2EE servers ensure that both databases are updated in the same transaction.

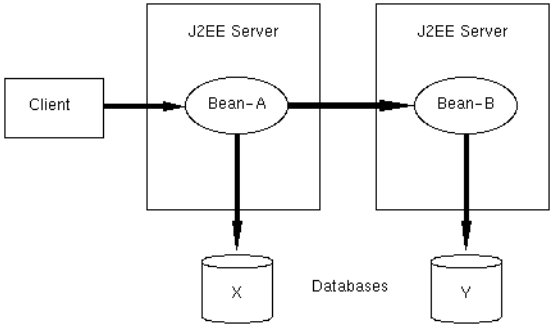

FIGURE 6-4 Updating Multiple Databases Across J2EE Servers

CHAPTER 7

Clients

The flexibility of the J2EE architecture allows enterprise beans to have a variety of clients:

- Stand-Alone Java™ Applications
- J2EE Application Clients
- Servlets
- JavaServer Pages™ Components
- Other Enterprise Beans

This chapter includes simple examples for each of these clients. For more complex examples, see the J2EE Blueprints web site (http://java.sun.com/j2ee/blueprints/index.html).

Stand-Alone Java™ Applications

Previous chapters demonstrate enterprise beans whose clients are stand-alone Java™ applications. The Getting Started chapter, for example, shows you how to create and run the `ConverterClient` program, a stand-alone Java application. The `ConverterClient` program is the client for the `ConverterApp` J2EE application. When you deploy `ConverterApp`, you select a checkbox labelled "Return client Jar" and then specify `ConverterAppClient.jar`. This file contains stub classes enabling the client to communicate with the enterprise bean. You include the `ConverterAppClient.jar` file in the classpath when running the `ConverterClient` program:

UNIX:

```
CPATH=$J2EE_HOME/lib/j2ee.jar:ConverterAppClient.jar:.
java -classpath "$CPATH" ConverterClient
```

Windows:

```
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar;ConverterAppClient.jar
java -classpath "%CPATH%" ConverterClient
```

For the previous commands to work, the `ConverterClient` and the J2EE server hosting the enterprise bean must run on the same machine. If a stand-alone Java application client is to run on a different machine, you must follow these steps:

1. Copy the client .jar file to the same machine where the client program will run.

2. Include the `-Dorg.omg.CORBA.ORBInitialHost` option when running the client. For example, if your enterprise bean has been deployed on a host named buzz, you would run the client as follows:

UNIX:

```
CPATH=.:$J2EE_HOME/lib/j2ee.jar:ConverterAppClient.jar
java -Dorg.omg.CORBA.ORBInitialHost=buzz -classpath "$CPATH"
ConverterClient
```

Windows:

```
CPATH=.;%J2EE_HOME%\lib\j2ee.jar;ConverterAppClient.jar
java -Dorg.omg.CORBA.ORBInitialHost=buzz -classpath "%CPATH%"
ConverterClient
```

J2EE Application Clients

Although a J2EE application client is a Java application, it differs from a stand-alone Java application client because it is a J2EE component. Like other J2EE components, a J2EE application client is created with the Application Deployment Tool and added to a J2EE application. Because it is part of a J2EE application, a J2EE application client has two advantages over a stand-alone Java application client. First, a J2EE application client is portable-- it will run on any J2EE-compliant server. Second, it may access J2EE services.

Accessing J2EE Services

A J2EE application client may use the following J2EE services:

- Security authentication
- JNDI lookups

Security Authentication

Of the J2EE services available to a J2EE application client, authentication is probably the most useful. Because the the authentication service provides a log-in mechanism, you don't have to code the log-in routines. When you run the J2EE application client, a log-in window automatically pops open and requests a J2EE user name and password. The authentication service verifies the user name and password before allowing the client to access the enterprise beans. For more information on enterprise bean security, see the Security chapter.

JNDI Lookups

A J2EE application client may use the JNDI API to look up enterprise beans, resources (databases), and environment entries. For example, the `J2EEClient` program locates an enterprise bean with the following call:

```
Object objref = initial.lookup("java:comp/env/ejb/SimpleConverter");
```

The `ejb/SimpleConverter` string is the name by which the `J2EEClient` code refers to the enterprise bean. This name does not have to be identical to the JNDI name of the enterprise bean in the deployed application. For example, the enterprise bean's JNDI name might be `MyConverter`. In this case, you would map `ejb/SimpleConverter`, the name coded in the client, to `MyConverter`, the JNDI name of the enterprise bean. (A later section "Specifying the JNDI Name" on page 99, shows you how to map the names with the Application Deployment Tool.)

Although this mapping adds a level of indirection, it provides a major benefit to distributed applications: Clients may use different names to refer to the same enterprise bean. The `J2EEClient` code, for example, uses the `SimpleConverter` name, but another client might refer to the same bean as `CurrencyConverter`. Even if you changed the bean's JNDI name (`MyConverter`) on the server, you wouldn't have to change the client's source code. But you would have to map the names again and re-deploy the application.

A stand-alone Java application client (such as `ConverterClient`), also locates an enterprise bean by calling the JNDI `lookup` method. However, the name coded in the stand-alone client must be identical to the bean's JNDI name on the server. You cannot map the two names in the Application Deployment Tool.

Setting Up the Application for the J2EE Application Client

The sections that follow show how to create and run an application with a J2EE application client called `J2EEClient`. The application contains the session bean documented in the Getting Started chapter. The source code required by the

application resides in the `doc/guides/ejb/examples/converter` subdirectory. Before stepping through the instructions in the next section, you should prepare the application:

- Create a J2EE application called `ConverterApp`.
- Create a session bean for the `ConverterEJB` class.
- Add this session bean to the J2EE application.
- Set the JNDI name for the session bean to `MyConverter`.
- Compile the `J2EEClient` program.

Creating the J2EE Application Client

You create a J2EE application client with the New Application Client Wizard of the Application Deployment tool. To start the wizard, from the File menu choose New Application client. The wizard displays the following dialog boxes. (You may skip any dialog box not listed here.)

Introduction Dialog Box:

a. Read this explanatory text for an overview of the wizard's features.

b. Click Next.

General Dialog Box:

a. Click the Add button next to the Contents text area.

b. In the dialog box titled "Add files to JAR," choose the directory containing the `J2EEClient.class` file (`examples/converter`). You may either type the directory name in the Root Directory field or locate it by clicking Browse.

c. Select the `J2EEClient.class` file from the text area and click Add.

d. Click OK.

e. In the Display Name field, enter `MyConverterClient`.

f. Click Next.

Enterprise Bean References Dialog Box:

a. Click Add.

b. In the Coded Name column enter `ejb/SimpleConverter`.

c. In the Type column select `Session`.

d. In the Home column enter `ConverterHome`.

e. In the Remote column enter `Converter`.

f. Click Finish.

Specifying the JNDI Name

In the `ConverterApp` application, the JNDI name of the `ConverterBean` is `MyConverter`. But the `J2EEClient` program refers to the same enterprise bean as `SimpleConverter`. Therefore, you must map the `SimpleConverter` name to `MyConverter`:

1. In the tree view, select the `ConverterApp` application.
2. In the JNDI Names tabbed pane perform these steps:
   a. Verify that the `ConverterBean` component has the `MyConverter` JNDI name.
   b. For the `ejb/SimpleConverter` reference, enter `MyConverter` for the JNDI name.

Deploying the J2EE Application

1. From the Tools menu, choose Deploy Application.
2. In the first dialog box, select the checkbox labelled “Return Client Jar.”
3. In the text field that appears, enter the full path name for the file `ConverterAppClient.jar`.

Running the J2EE Application Client

To run the `J2EEClient` program , enter this command:

```
runclient -client ConverterApp.ear -name MyConverterClient
```

The `ConverterApp.ear` file contains the J2EE application. The `MyConverterClient` option is the display name of the J2EE application client component.

The client container pops open a login window. In the User Name field enter `guest` and in the Password field enter `guest123`. (These entries are the default values specified in the `config/auth.properties` file.)

If the J2EE application client does not reside on the same host as the J2EE server, you should follow these steps:

1. Copy the `ConverterAppClient.jar` file to the client host machine. Created during deployment, the `ConverterAppClient.jar` file contains stub classes that enable remote connectivity.

2. Copy the `ConverterApp.ear` file to the client host machine.

3. Set the APPCPATH environment variable to the fully qualified name of the .jar file. In the following example, the fully qualified name is `/user/duke/ConverterAppClient.jar`:

UNIX, C-shell:

```
setenv APPCPATH /user/duke/ConverterAppClient.jar
```

UNIX, Bourne and Korn shells:

```
APPCPATH=/user/duke/ConverterAppClient.jar
export APPCPATH
```

Windows:

```
set APPCPATH=\user\duke\ConverterAppClient.jar
```

4. Set the VMARGS environment variable to the `-Dorg.omg.CORBA.ORBInitialHost` option. In the following example, the option points to the remote host named buzz, where the J2EE server is running:

UNIX, C-shell:

```
setenv VMARGS -Dorg.omg.CORBA.ORBInitialHost=buzz
```

UNIX, Bourne and Korn shells:

```
VMARGS="-Dorg.omg.CORBA.ORBInitialHost=buzz"
export VMARGS
```

Windows:

```
set VMARGS=-Dorg.omg.CORBA.ORBInitialHost=buzz
```

5. Run the `runclient` script:

```
runclient -client ConverterApp.ear -name MyConverterClient
```

Servlets

Servlet clients allow web browsers to indirectly access enterprise beans. The following diagram illustrates this access:

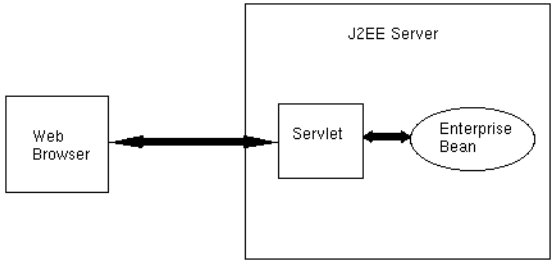

FIGURE 7-1 Servlet Client of an Enterprise Bean

The sections that follow show how to create a servlet client called `AdderServlet`. This servlet accesses an enterprise bean implemented by the `AdderEJB` class. The enterprise bean maintains a running total that is incremented by a value entered by the end-user in a browser. The source code for `AdderServlet` and `AdderEJB` reside in the `doc/guides/ejb/examples/adder` directory.

Setting Up the Servlet's J2EE Application

The J2EE application in this example contains an enterprise bean and a web component. The enterprise bean is packaged in an EJB .jar file and the web component in a .war file. A later section ("Creating the Servlet's .war File" on page 104) explains how to package the `AdderServlet.class` and `adder.html` files. But first, you should prepare the J2EE application:

- Create a J2EE application called `AdderApp`.
- Create a session bean for the `AdderEJB` class.
- Add this session bean to the J2EE application.
- Set the display name for the session bean to `AdderBean`.

Coding the Servlet

This section briefly describes the `AdderServlet` example. If you need additional information about servlets, please visit the Java™ Servlet API web page (java.sun.com / products / servlet / index.html).

Immediately after it creates the `AdderServlet` instance, the web server calls intance's `init` method. This method looks up the enterprise bean, whose coded name is `ejb/Adder`, and then creates the bean:

```
public void init() throws ServletException {
```

```
    try {
        InitialContext ic = new InitialContext();
        Object objref = ic.lookup("java:comp/env/ejb/Adder");
        AdderHome home =
           (AdderHome)PortableRemoteObject.narrow(objref,
                                        AdderHome.class);
        adder = home.create(0);
    } catch(Exception e) {
         e.printStackTrace();
    }

}
```

When the end-user clicks the Submit Query button on the HTML form, the web server calls the `doGet` method of the `AdderServlet`. The `doGet` method fetches the `inputString` entered by the end-user, converts it an integer, and adds it to the running total in the enterprise bean by invoking the `add` business method. To retrieve the running total, the `doGet` method calls the `getTotal` business method. The source code for the `doGet` method follows:

```
public void doGet (HttpServletRequest req, HttpServletResponse res)
                  throws ServletException, IOException {

   String inputString = req.getParameter("inputString");
   Integer inputNumber = new Integer(inputString);
   adder.add(inputNumber.intValue());
   int total = adder.getTotal();
   res.setContentType("text/html");
   PrintWriter out = res.getWriter();
   generatePage(out,total);

}
```

The `generatePage` method, called by `doGet`, formats the HTML page that displays the running total:

```
private void generatePage(PrintWriter out, int total) {

   out.println("<html>");
   out.println("<head>");
```

```
    out.println(“<title>Input for AdderServlet</title>”);
    out.println(“</head>”);
    out.println(“<body>”);
    out.println(“The running total is: “ + String.valueOf(total));
    out.println(“<p>”);
    out.println(“<form method = get action=\”AdderAlias\”>”);
    out.println(“Please enter an integer:”);
    out.println(“<input type=text name=\”inputString\”>”);
    out.println(“<p>”);
    out.println(“<input type=submit>”);
    out.println(“</form>”);
    out.println(“</body>”);
    out.println(“</html>”);
}
```

Compiling the Servlet

When compiling `AdderServlet.java`, you must include the `j2ee.jar` file in the classpath:

UNIX:

```
CPATH=.:$J2EE_HOME/lib/j2ee.jar
javac -classpath “$CPATH” AdderServlet.java
```

Windows:

```
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar
javac -classpath %CPATH% AdderServlet.java
```

Coding the HTML File

The `adder.html` file displays a form that prompts the user for input. When the user clicks the form’s Submit button, the server invokes the `AdderServlet` class, which you will map to the `AdderAlias` in the next section. Here is the content of the `adder.html` file:

```
<html>
<head>
<title>Initial Page for AdderServlet</title>
```

```
</head>
<body>
<form method = get action="AdderAlias">
Please enter an integer:
<input type=text name="inputString">
<p>
<input type=submit>
</form>
</body>
</html>
```

Creating the Servlet's .war File

To package the `AdderServlet.class` and `adder.html` files in a .war file, you run the New Web Component Wizard of the Application Deployment Tool. To start the wizard, from the File menu choose New Web Component. The wizard displays the following dialog boxes. (You may skip any dialog boxes not listed here.)

WAR File General Properties Dialog Box:

a. In the combo box labelled "Web Component Will Go In," select AdderApp.

b. In the WAR Display Name field, enter `AdderWAR`.

c. Click Add.

d. In the Add Content Files dialog box, choose the root directory containing the `adder.html` file (`examples/adder`). You may either type the directory name in the Root Directory field or locate it by clicking Browse.

e. Select the `adder.html` file from the text area and click Add.

f. Click Next.

g. In the Add Class Files dialog box choose, choose the `examples/adder` directory again.

h. Select the `AdderServlet.class` file from the text area and click Add.

i. Click Finish.

j. Click Next.

Choose Component Type Dialog Box:

a. Select Servlet.

b. Click Next.

Component General Properties Dialog Box:

a. In the Servlet Class combo box, select AdderServlet.

b. In the Web Component Display Name field, enter `TheAdder`.

c. Click Next.

Component Aliases Dialog Box:

a. Click Add.

b. In the Aliases list, enter `AdderAlias`.

c. Click Next.

Enterprise Bean References Dialog Box:

a. Click Add.

b. In the Coded Name column enter `ejb/Adder`.

c. In the Type column select `Session`.

d. In the Home column enter `AdderHome`.

e. In the Remote column enter `Adder`.

f. Click Finish.

Specifying the Web Context Root

1. In the tree view select AdderApp.

2. In the Web Context tabbed pane, enter `AdderContextRoot` in the ContextRoot column.

Specifying the JNDI Names

In the JNDI Names tabbed pane for the `AdderApp`, specify `MyAdder` as the JNDI name for both the `ejb/Adder` reference and the `AdderBean` component.

Deploying the Servlet's J2EE Application

1. From the Tools menu, choose Deploy Application.
2. In the first dialog box, do not select the checkbox labelled "Return Client Jar."
3. In the second dialog box, verify the JNDI names.
4. In the third dialog box, verify the context root.

Running the Servlet

To run the `AdderServlet` program from your browser, specify the URL as follows, but replace `<host>` with the name of the machine that is running the J2EE server:

```
http://<host>:8000/AdderContextRoot/adder.html
```

Enter an integer in the field and click the Submit Query button. Repeat this process and note the running total displayed at the top of the page.

JavaServer Pages™ Components

A JavaServer Pages™ (JSP) component may use a JavaBeans™ component as a proxy to access an enterprise bean. The following diagram illustrates how these components work together:

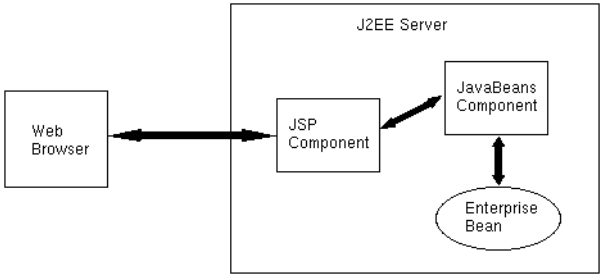

FIGURE 7-2 JSP Client of an Enterprise Bean

The sections that follow show how to create a J2EE application with the following elements:

- JSP client: `Account.jsp`

- JavaBeans component: `AccountBean`
- Entity bean: `AccountEJB`

The `Account.jsp` and `AccountBean.java` files are in the `doc/guides/ejb/examples/jsptobean` directory. The `AccountEJB.java` source code is in the `doc/guides/ejb/examples/account` directory.

Setting Up the JSP Component's J2EE Application

The following figure shows the J2EE application that contains the JSP component. Stored in a .ear file, the J2EE application holds an EJB .jar file and a web component .war file. The EJB .jar file contains the `AccountEJB` entity bean described in the section, "A Bean-Managed Persistence Example" on page 44. The .war file contains the JSP component (`Account.jsp`) and the JavaBeans component (`AccountBean`).

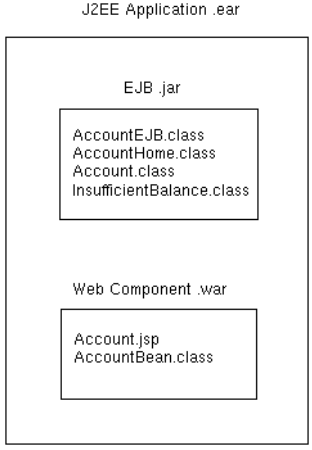

FIGURE 7-3 J2EE Application .ear File

The sections that follow describe how to create and package the web components, but first, you should set up the J2EE application:

- Create a J2EE application named `AccountJSPApp`.
- Create an entity bean for the `AccountEJB` class.
- Add the entity bean to the J2EE application.
- Set the display name for the entity bean to `AccountBean`.

- Create the ACCOUNT database table.

If you need to review the instructions for performing the preceding tasks, see the section "Tips on Running the AccountEJB Example" on page 56.

Writing the JSP File

This section briefly describes the JSP tags in the `Account.jsp` file. These tags are marked by bold font in the full listing of `Account.jsp` included at the end of this section. (For more information on writing JSP files, see the JavaServer Pages web site at java.sun.com/products/jsp/index.html.)

The first JSP tag in the `Account.jsp` file specifies the JavaBeans component. The `jsp:useBean` tag creates a JavaBeans component by instantiating the `AccountBean` class, names the component `accountBean`, and indicates that the component will be available for the current HTTP session. Unlike the other JSP tags, the `jsp:useBean` tag is executed just once during the session-- when the end-user first accesses the `Account.jsp` page:

```
<jsp:useBean id="accountBean" scope="session" class="AccountBean" />
```

The `jsp:setProperty` tag sets all of the property values in the `accountBean` to the parameters passed by the HTML form. For example, if the end-user selects the Debit radio button and clicks Submit, the `action` parameter is set to `debit`. This parameter value is sent to the web server, which assigns it to the `action` property value of the `accountBean`. The asterisk in the tag indicates that all properties will be set:

```
<jsp:setProperty name="accountBean" property="*" />
```

JSP scripting elements are enclosed as follows: <% *element* %>. The first scripting element declares the `status` variable:

```
<%! String status; %>
```

The next scripting element invokes `processRequest` method of the `accountBean` object. This method checks the `action` property value and invokes methods on the entity bean:

```
<% status = accountBean.processRequest(); %>
```

Near the bottom of the `Account.jsp` file, the `status` variable returned by the `processRequest` method is displayed:

```
<h3><b>Status :</b></h3> <%= status %>
```

In the HTML `form` tag, the `action` parameter indicates that the `Account.jsp` page is to be executed when the end-user clicks the form's Submit button. (Behind the scenes, the web server transforms the `Account.jsp` page into a servlet which it then executes.) Except for the `jsp:useBean` tag, every JSP tag and scripting element in the `Account.jsp` page is executed whenever the end-user clicks Submit:

```
<form method=POST action=Account.jsp>
```

Each `jsp:getProperty` tag fetches a property value from the `accountBean`. For example, the second `jsp:getProperty` tag retrieves the `balance` property from the `accountBean`. The `balance` is displayed in the text field of the HTML form. The next time the end-user clicks the Submit button, the value in the text field is sent to the web server as the `balance` parameter. The `jsp:setProperty` tag causes the web server to assign this parameter to the `balance` property of the `accountBean`. Here is the `jsp:getProperty` tag for the `balance` property:

```
<INPUT type=text name="balance"  size="8"
value="<jsp:getProperty name="accountBean" property="balance" />">
```

The full listing for the `Account.jsp` file follows:

```
<html>
<jsp:useBean id="accountBean" scope="session" class="AccountBean" />
<jsp:setProperty name="accountBean" property="*" />
<%! String status; %>

<% status = accountBean.processRequest(); %>

<html>
<head>
    <title>Account JSP</title>
</head>
<body background="back.gif">
<font size = 5 color="#CC0000">
<h1><b><center>Account JSP Example</center></b></h1>
<hr>
<br>
<form method=POST action=Account.jsp>
<br>

<br>Account ID
<INPUT type=text name="id" size="8"
value="<jsp:getProperty name="accountBean" property="id" />" >
```

```
Balance
<INPUT type=text name=”balance”  size=”8”
value=”<jsp:getProperty name=”accountBean” property=”balance” />” >
<br>
First Name
<INPUT type=text name=”firstName” size=”8”
value=”<jsp:getProperty name=”accountBean” property=”firstName” />”>
Last Name
<INPUT type=text name=”lastName” “size=8”
Value=”<jsp:getProperty name=”accountBean” property=”lastName” />”
><br>

<br>
<h2><b>Action :</b></h2>
  <INPUT type=”radio” name=”action” value=”find”>Find
  <INPUT type=”radio” name=”action” value=”create”>Create
  <INPUT type=”radio” name=”action” value=”debit”>Debit
  <INPUT type=”radio” name=”action” value=”credit”>Credit
<br>

<br>Amount  <INPUT type=text name=”amount”><br>
<INPUT type=submit name=”submit” value=”Submit”>

</form>
</FONT>
</body>
</html>

<hr>
<h3><b>Status :</b></h3> <%= status %>
</html>
```

Coding the JavaBeans Component

The following description of the AccountBean code is quite brief. For more information on coding JavaBeans components , see the JavaBeans home page (java.sun.com / products / beans / index.html).

The JavaBeans component is created when the end-user first accesses the Account.jsp page. (This action is specified by the jsp:useBean tag in the Account.jsp file.) The AccountBean class accesses the entity bean implemented by the AccountEJB class. The constructor of the AccountBean class locates the entity bean's home interface by invoking the lookup method:

```
public AccountBean() {


   try {
       Context ic = new InitialContext();
       java.lang.Object objref =
          ic.lookup("java:comp/env/ejb/Account");
       accountHome =
           (AccountHome) PortableRemoteObject.narrow(objref,
                                          AccountHome.class);
   } catch (Exception re) {
       System.err.println ("Couldn't locate Account Home");
       re.printStackTrace();
   }
   reset();
}
```

The end-user indicates the action parameter by selecting a radio button on the Account.jsp page. The web server assigns the parameter value to the action property of the AccountBean. Invoked by a scripting element in the Account.jsp page, the processRequest method of the AccountBean checks the value of the action property. If the action property is create, for example, the processRequest method creates a new entity bean. Here is the code for the processRequest method:

```
public String processRequest() {


   String  message = "";


   System.out.println("Process request called ");
   System.out.println(this);
```

```
try {

  if( action.equals(“create”) ) {

     account =
       accountHome.create(id, firstName, lastName, balance);
     message = “Created account ‘” + id + “‘”;

  }
  else if( action.equals(“debit”) ) {

     account = accountHome.findByPrimaryKey(id);
     account.debit(amount);
     loadFromEJB();
     message = “Debited account ‘” + id + “‘ by $” + amount;

  }
  else if( action.equals(“credit”) ) {

     account = accountHome.findByPrimaryKey(id);
     account.credit(amount);
     loadFromEJB();
     message = “Credited account ‘” + id + “‘ by $” + amount;

  }
  else if( action.equals(“find”) ) {

     account = accountHome.findByPrimaryKey(id);
     loadFromEJB();
     message = “Found account ‘” + id;

  }

} // try
```

```
    catch (Exception e) {
      message = e.toString();
    }

    return message;
}
```

The state of the AccountBean instance mirrors that of the entity bean instance. Both instances contain these variables: id, firstName, lastName, and balance. The AccountBean changes the balance variable of the entity bean by invoking the debit and credit business methods. To refresh its state from the entity bean, the AccountBean invokes business methods such as getFirstName and getLastName. It invokes these getter methods in the private method loadFromEJB:

```
private void loadFromEJB()
{
    System.out.println("Calling loadFromEJB()");
    try {
        setFirstName(account.getFirstName());
        setLastName(account.getLastName());
        setBalance(account.getBalance());
     } catch (Exception re) {
         System.err.println
            ("Failed to load AccountBean from AccountEJB.");
         re.printStackTrace();
     }
}
```

Compiling the JavaBeans Component

When compiling AccountBean.java, you must include the j2ee.jar and ejb.jar files in the classpath. The ejb.jar file is required because it contains the class files for the Account and AccountHome interfaces-- types used by the AccountBean class. When you created the enterprise bean (AccountEJB) for the AccountJSPApp application, the tool inserted the ejb.jar file into the AccountJSPApp.ear file. To extract the ejb.jar file from the AccountJSPApp.ear file, follow these steps:

1. In the tree view, select the EJB .jar file for the AccountJSPApp application.

2. From the File menu, choose Save As.

3. Save the `ejb.jar` file in the `examples/jsptobean` directory.

To compile the JavaBeans component, change to the `examples/jsptobean` directory and execute these commands:

UNIX:

```
CPATH=.:$J2EE_HOME/lib/j2ee.jar:ejb.jar
javac -classpath "$CPATH" AccountBean.java
```

Windows:

```
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar;ejb.jar
javac -classpath %CPATH% AccountBean.java
```

Creating the JSP Component's .war File

To package the `AccountBean.class` and `Account.jsp` files in a .war file, you run the New Web Component Wizard of the Application Deployment Tool. To start the wizard, from the File menu choose New Web Component. The wizard displays the following dialog boxes. (You may skip any dialog boxes not listed here.)

WAR File General Properties Dialog Box:

a. In the combo box labelled "Web Component Will Go In," select AccountJSPApp.

b. In the WAR Display Name field, enter `AccountWAR`.

c. Click Add.

d. In the Add Content Files dialog box, choose the root directory containing the `Account.jsp` file (`examples/jsptobean`). You may either type the directory name in the Root Directory field or locate it by clicking Browse.

e. Select the `Account.jsp` file from the text area and click Add.

f. Click Next.

g. In the Add Class Files dialog box choose, the `examples/jsptobean` directory again.

h. Select the `AccountBean.class` file from the text area and click Add.

i. Click Finish.

j. Click Next.

Choose Component Type Dialog Box:

a. Select JSP.

b. Click Next.

Component General Properties Dialog Box:

a. In the JSP Filename combo box, select Account.jsp.

b. In the Web Component Display Name field, enter `TheAccount`.

c. Click Next.

Enterprise Bean References Dialog Box:

a. Click Add.

b. In the Coded Name column enter `ejb/Account`.

c. In the Type column select `Entity`.

d. In the Home column enter `AccountHome`.

e. In the Remote column enter `Account`.

f. Click Finish.

Specifying the Web Context Root

1. In the tree view select AccountJSPApp.

2. In the Web Context tabbed pane, enter `AccountContextRoot` in the ContextRoot column.

Specifying the JNDI Names

In the JNDI Names tabbed pane for the `AccountJSPApp`, enter the JNDI names shown in the following table:

TABLE 7-1 AccountJSPApp JNDI Names

Component/ Reference Name	JNDI Name
AccountBean	MyAccount
jdbc/AccountDB	jdbc/Cloudscape
ejb/Account	MyAccount

Deploying the JSP Component's J2EE Application

1. From the Tools menu, choose Deploy Application.
2. In the first dialog box, do not select the checkbox labelled "Return Client Jar."
3. In the second dialog box, verify the JNDI names.
4. In the third dialog box, verify the context root.

Running the JSP Component

To run `Account.jsp` from your browser, specify the URL as follows, but replace `<host>` with the name of the machine that is running the J2EE server:

```
http://<host>:8000/AccountContextRoot/Account.jsp
```

To create a new account, follow these steps:

1. In the Account ID field, enter a three-digit integer.
2. In the Balance field, enter the intitial balance (for example, 100.00).
3. In the First Name and Last Name fields enter your name.
4. Under Action, select Create.
5. Click the Submit button.

To credit an account, perform the following tasks:

1. In the Account ID field, enter the three-digit account identifier.
2. In the Amount field, enter the credit amount (for example, 55.00).
3. Under Action, select Credit.

5. Click the Submit button.

Other Enterprise Beans

Although enterprise beans run on the server, one enterprise bean may be the client of another. For example, a session bean is often the client of an entity bean. And the session bean has clients of its own-- perhaps JSP components or Java applications.

The following figure shows the structure of a J2EE application in which one enterprise bean is the client of another. The `ShipperClient` is a stand-alone Java application that accesses the `Shipper` session bean. The `Shipper` session bean is the client of the `Stock` entity bean, which accesses a database.

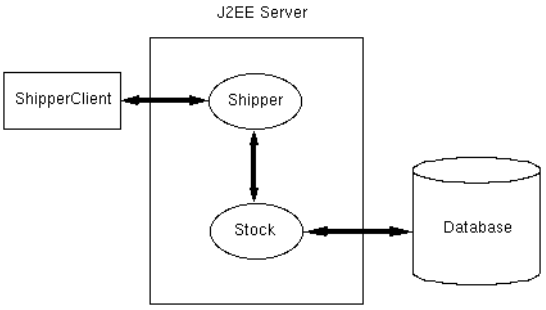

FIGURE 7-4 The `ShipperApp` Sample Application

Setting Up the ShipperApp Application

The sections that follow explain how to assemble the `ShipperApp` application illustrated in figure 7-4. Before proceeding, you should perform these tasks:

1. Compile the source code files in the `doc/guides/ejb/examples/shipper` directory:

- `StockEJB.java`
- `StockHome.java`
- `Stock.java`
- `ShipperEJB.java`
- `ShipperHome.java`
- `Shipper.java`

- `ShipperClient.java`

2. With the Application Deployment Tool, create a J2EE application named `ShipperApp`.

Creating the Stock EJB .jar File

Invoke the New Enterprise Bean Wizard and step through the following dialog boxes. (You may skip any dialog box not listed here.)

EJB JAR Dialog Box:

a. In the combo box labelled "Enterprise Bean will go in," select ShipperApp.

b. In the JAR Display Name field, enter `StockJAR`.

c. Add the following files to the Contents text area: `Stock.class`, `StockEJB.class`, `StockHome.class`.

General Dialog Box:

a. In the Enterprise Bean Class combo box, select StockEJB.

b. In the Home Interface combo box, select StockHome.

c. In the Remote Interface combo box, select Stock.

d. Select the Entity radio button.

e. In the Enterprise Bean Display Name field, enter `StockBean`.

Entity Settings Dialog Box:

Select the radio button labelled "Bean-Managed Persistence."

Resource References Dialog Box:

a. In the Coded Name column enter `jdbc/StockDB`.

b. In the other columns retain the default values.

Transaction Management Dialog Box:

a. In the Transaction Type column, select Required for the `updateOnHand` and `reorderNeeded` methods.

b. Click Finish

Creating the Shipper EJB .jar File

Invoke the New Enterprise Bean Wizard and step through the following dialog boxes. (You may skip any dialog box not listed here.)

EJB JAR Dialog Box:

a. In the combo box labelled “Enterprise Bean will go in,” select ShipperApp.

b. In the JAR Display Name field, enter `ShipperJAR`.

c. Add the following files to the Contents text area: `Shipper.class`, `ShipperEJB.class`, `ShipperHome.class`.

General Dialog Box:

a. In the Enterprise Bean Class combo box, select ShipperEJB.

b. In the Home Interface combo box, select ShipperHome.

c. In the Remote Interface combo box, select Shipper.

d. Select the Session radio button.

e. Select the Stateful radio button.

f. In the Enterprise Bean Display Name field, enter `ShipperBean`.

Enterprise Bean References Dialog Box:

a. Click Add.

b. In the Coded Name column enter `ejb/Stocker`.

c. In the Type column select `Entity`.

d. In the Home column enter `StockHome`.

e. In the Remote column enter `Stock`.

f. Click Finish.

Specifying the JNDI Names

In the JNDI Names tabbed pane for the `ShipperApp`, enter the JNDI names shown in the following table:

TABLE 7-2 ShipperApp JNDI Names

Component/ Reference Name	JNDI Name
ejb/Stocker	MyStocker
jdbc/StockDB	jdbc/Cloudscape
ShipperBean	MyShipper
StockBean	MyStocker

The first row in the table links the `ejb/Stocker` reference to the `MyStocker` JNDI name. The `ShipperEJB` class uses `ejb/Stocker` to reference the `StockBean` in the following call:

```
Object objref = initial.lookup("java:comp/env/ejb/Stocker");
```

The `MyStocker` JNDI name corresponds to `StockBean`, the entity bean component for the `StockEJB` class. This class fetches a `DataSource` so that it can connect to a Cloudscape database:

```
private String dbName = "java:comp/env/jdbc/StockDB";
DataSource ds = (DataSource) ic.lookup(dbName);
```

`MyShipper` is the JNDI name for `ShipperBean`, the session bean component for the `SessionEJB` class. The `SessionClient` program locates the `ShipperBean` with the following call:

```
Object objref = initial.lookup("MyShipper");
```

Deploying and Running the J2EE Application

1. When you deploy the `ShipperApp`, select the checkbox labelled “Return Client Jar”. In the text field that appears enter the full path name for the `ShipperAppClient.jar` file.

2. Create the database table used by the `ShipperApp`:

UNIX:

```
cd $J2EE_HOME/doc/guides/ejb/examples/shipper
../util/cloudTable.sh
```

Windows:

```
cd %J2EE_HOME%\doc\guides\ejb\examples\shipper
..\util\cloudTable
```

3. To try out the `ShipperApp`, run the client program:

UNIX:

```
CPATH=$J2EE_HOME/lib/j2ee.jar:ShipperAppClient.jar:.
java  -classpath "$CPATH" ShipperClient
```

Windows:

```
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar;ShipperAppClient.jar
java -classpath "%CPATH%" ShipperClient
```

4. Although the client program does not print any results, there are two ways you can verify that the `ShipperApp` ran correctly:

a. If you started the J2EE server with the `-verbose` option, it will print these lines:

```
ShipperEJB: ejbCreate()
ShipperEJB: shipItem()
StockEJB: updateOnHand()
StockEJB: reorderNeeded()
```

b. Run the `cloudIJ.sh` (UNIX) or `cloudIJ.bat` (Windows) script in the `examples/util` directory and query the STOCK table in the database:

```
ij> select * from stock;
PR&|QUANTITY_ON_HA&|REORDER_LEVEL
-----------------------------------
123|91            |90

1 row selected
```

CHAPTER 8

Security

You declare the security attributes of an enterprise bean in with the Application Deployment Tool. This declarative approach to security enforcement has two major advantages. First, you save time because you don't have to code and debug security routines in your enterprise beans or their clients. Second, the administrator of the J2EE server can customize the security attributes for a particular production environment at deployment time.

The J2EE server enforces security at two levels: Authentication and Authorization.

Authentication

Authentication is the process by which a user proves his or her identity to a system. For example, when you log on to a computer and provide a password, the software that verifies your user name and password is performing authentication. The J2EE server controls client access with a distributed authentication service. This service controls whether or not a J2EE user can access the components within a J2EE application.

Note: This section describes the authentication service of the J2EE SDK. Other J2EE implementations might perform authentication differently. In a commercial implementation of J2EE, for example, a J2EE user and an operating system user might be the same, but in the J2EE SDK they are not.

J2EE Users, Realms, and Groups

A J2EE user is similar to an operating system user. Typically, both types of users represent people. However, these two types of users are not the same. The J2EE authentication service has no knowledge of the user and password you provide

when logging on to the operating system. The J2EE authentication service is not connected to the security mechanism of the operating system. The two security services manage users that belong to different realms.

A *realm* is a collection of users that are controlled by the same authentication policy. The J2EE authentication service governs users in two realms: certificate and default.

Certificates are used with the HTTPS protocol to authenticate Web browser clients. (For more information on certificates, see the Security in JDK 1.2chapter of the Java™ Tutorial at java.sun.com/docs/books/tutorial/security1.2/TOC.html.) To verify the identity of a user in the certificate realm, the authentication service verifies a X509 certificate. (For step-by-step instructions, see the Setting Up a Server Certificate section.) The common name field of the X509 certificate is used as the principal name.

In most cases, the J2EE authentication service verifies user identity by checking the default realm. This realm is used for the authentication of all clients except for Web browser clients that use the HTTPS protocol and certificates.

A J2EE user of the default realm may belong to J2EE group. (A user in the certificate realm may not.) A *group* is a category of users, classified by common traits such as job title or customer profile. For example, most customers of an e-commerce application might belong to the CUSTOMER group, but the big spenders would belong to the PREFERRED group. Categorizing users into groups makes it easier to control the access of large numbers of users. A later section, Authorization, discusses controlling user access to enterprise beans.

Client Authentication

The J2EE authentication service controls access from all types of bean clients: J2EE application clients, stand-alone Java applications, and web components.

When a J2EE application client starts running, its container pops open a window that requests the J2EE user name and password. If you run the `J2EEClient` program of the Clients chapter (J2EE Application Clients section), you'll see this log-on window in action. The authentication service verifies that the user name and password from the log-on window exist in the default realm. After authentication, the user's security context is associated with any call that the client makes to enterprise beans deployed in the J2EE server.

Most of the examples in this book feature clients that are stand-alone Java applications. Because these clients do not log on, they are assigned the unauthenticated and anonymous user named `guest`. (The password is `guest123`.) Other types of clients, including Web browsers, may also access the J2EE server without authentication. Such clients are always assigned the user `guest`, indicating that their access in unauthenticated.

Many applications do not require authentication. For example, an online product catalog would not force customers to log on if they are merely browsing. Also, when you first start developing an application, you may find it convenient to allow anyone (`guest`) to access the application's components.

During deployment, you specify whether or not a web component is a protected resource. If the web component is unprotected, anyone may access it from their browser. If an unprotected web component accesses an enterprise bean, the authentication service assigns it a certificate for the `guest` user. Any subsequent calls to enterprise beans are associated with the `guest` user.

If a web component is protected, you may specify three types of authentication: basic, form, and certificate. With basic authentication, the server instructs the Web browser to prompt for the user name and password. With form authentication, you can specify the .html form or .jsp file that prompts for the user name and security:passwordspassword. With certificate authentication, the server requests a certificate from the browser. In all types of authentication, if the web component calls as enterprise bean, the call is associated with the authenticated user.

Managing J2EE Users and Groups

The `realmtool` utility is a command-line program that allows you to add and remove users in the default and certificate realms.

To display all users in the default realm, type this command:

```
realmtool -list default
```

To add a user to the default realm you specify the `-add` flag. The following command will add a user named robin who is protected by the password red, and will include robin in the bird and wing groups:

```
realmtool -add robin red bird,wing
```

To add a user to the certificate realm, you import a file containing the X509 certificate that identifies the user:

```
realmtool -import certificate-file
```

To remove a user you specify the `-remove` flag. For example, to remove a user named sparrow from the default realm, you would type the following command:

```
realmtool -remove default sparrow
```

To add a group to the default realm you specify the `-addGroup` flag. The following command adds the wing group:

```
realmtool -addGroup wing
```

(You cannot add a group to the certificate realm.)

To remove a group from the default realm, you specify the `-removeGroup` flag:

```
realmtool -removeGroup wing
```

Authorization

Authorization is the process by which the J2EE server grants or denies permission to invoke the methods of an enterprise bean. You define authorization in the enterprise bean's security attributes in three steps:

- Declaring Roles
- Declaring Method Permissions
- Mapping Roles to J2EE Users and Groups

Declaring Roles

When you design an enterprise bean, you should keep in mind what types of users will access the bean. For example, an Account enterprise bean might be accessed by customers, bank tellers, and branch managers. Each of these user categories is called a role.

A J2EE group also represents a category of users, but it has a different scope than a role. A J2EE group is designated for the entire J2EE server, whereas a role covers only a specific application in a J2EE server.

To create a role for an application, you declare it for the EJB .jar or web component (.war) files contained in the application. For example, to create a role for an enterprise bean, follow this procedure in the Application Deployment Tool:

1. In the tree view, select the enterprise bean's EJB .jar file.
2. In the Roles tabbed pane, click Add.
3. In the table, enter values for the Name and Description fields.

Declaring Method Permissions

After you've defined the roles, you're ready to define the method permissions of an enterprise bean. Method permissions indicate which roles are allowed to invoke which methods.

The following table shows how you might define the method permissions of an Account bean. Managers and tellers may create and remove accounts. Only managers are allowed to audit accounts. Customers may credit and debit their accounts and may transfer funds. In the table, an “X” indicates that the role may invoke the method, and a “0” indicates that permission is denied.

TABLE 8-1 Method Permissions in an Account Bean

Method Name	Manager	Teller	Customer
create	X	X	0
remove	X	X	0
audit	X	0	0
credit	0	0	X
debit	0	0	X
transfer	0	0	X

You specify method permissions by mapping roles to methods with the Application Deployment Tool:

1. In the tree view, select the enterprise bean.
2. Select the Security tabbed pane.
3. In the Method Permissions table, select a role’s checkbox if that role should be allowed to invoke a method.

Mapping Roles to J2EE Users and Groups

When you are developing an enterprise bean, you should know the roles of your users, but you probably won’t know exactly who the users will be. That’s okay, because after your bean has been deployed the administrator of the J2EE server will map the roles to the J2EE users (or groups) of the default realm. In the Account bean example, the adminstrator might assign the user Sally to the Manager role, and the users Bob, Ted, and Clara to the Teller role.

Using the Application Deployment Tool, the administrator maps roles to J2EE users and groups by following these steps:

1. In the tree view, select the application.
2. In the Security tabbed pane, select the appropriate role from the Role Name list.

3. Click Add.

4. In the Users dialog box, select the users and groups that should belong to the role. (The users and groups were created with the command-line `realmtool`.)

By default, the Role Name table assigns the ANYONE role to a method. The `guest` user, which is anonymous and unauthenticated, belongs to the ANYONE role. Therefore, if you do not map the roles, any user may invoke the methods of an enterprise bean.

Scenarios

The scenarios in this section show how authentication and authorization work together to manage security for J2EE applications.

J2EE Application Client

In this scenario, an employee named Bob has moved and he wishes to update his home address for his company's records. The company that Bob works for has a J2EE application that allows employees to update their personal information. Figure 8-1 illustrates this application. To change his address, Bob runs a J2EE application client that invokes the `update` method in the `Employee` enterprise bean.

Before Bob runs the client, the J2EE administrator sets up the security as follows:

- Only the Administrator and RegularEmployee roles may invoke the `update` method of the `Employee` enterprise bean.
- The J2EE group named FullEmployee belongs to the RegularEmployee role.
- The J2EE user Bob belongs to the FullEmployee group in the default realm.

The J2EE server performs the following security checks at run time:

1. When the J2EE application client starts running it opens a dialog that prompts for the J2EE user name and password, which Bob enters.

2. The authentication service verifies that Bob's user name and password exist in the default realm.

3. Bob clicks the update button in the client, which attempts to invoke the `update` method of the `Employee` enterprise bean.

4. The EJB container performs authorization. It verifies that the RegularEmployee role, to which Bob's group (FullEmployee) belongs, has permission to invoke the `update` method.

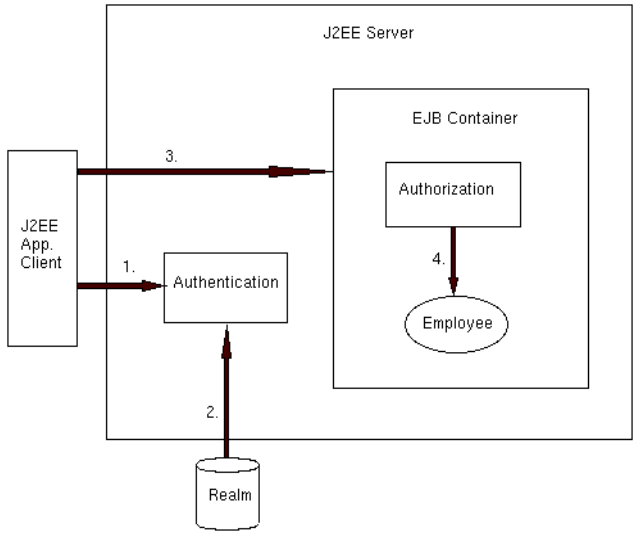

FIGURE 8-1 Authenticated Access to an Enteprise Bean

Web Browser Client

In the next scenario, illustrated in figure 8-2, Mary transfers money between her savings and checking accounts from her Web browser. To transfer the funds, Mary enters a URL that accesses a JSP component. This component calls a JavaBeans™ component, which invokes the `transfer` method of the `Account` enterprise bean.

The J2EE administrator enforces security with these rules:

- The JSP component is a protected resource.
- Only the Customer role may invoke the `transfer` method of the `Account` enterprise bean.
- The J2EE group named CurrentCustomer belongs to the Customer role.
- Mary's J2EE user belongs to the CurrentCustomer group in the default realm.

When Mary transfers the funds, the J2EE server enforces security as follows:

1. Mary's browser attempts to access the JSP component.

2. Because the component is a protected resource, authentication is required. The Web service requests the Web browser to prompt for the J2EE user name and password.

3. Mary enters her J2EE user name and password, which are passed back to the J2EE server.

4. The authentication service verifies that the user name and password exist in the default realm.

5. The Web browser is allowed to access the JSP component.

6. Mary clicks the Transfer button on the form generated by the JSP component, which calls a JavaBeans component.

7. The JavaBeans component attempts to invoke the `transfer` method of the `Account` enterprise bean.

8. Mary's J2EE group (CurrentCustomer) belongs to the Customer role, which is allowed to invoke the `transfer` method. Therefore, the EJB container authorizes the invocation.

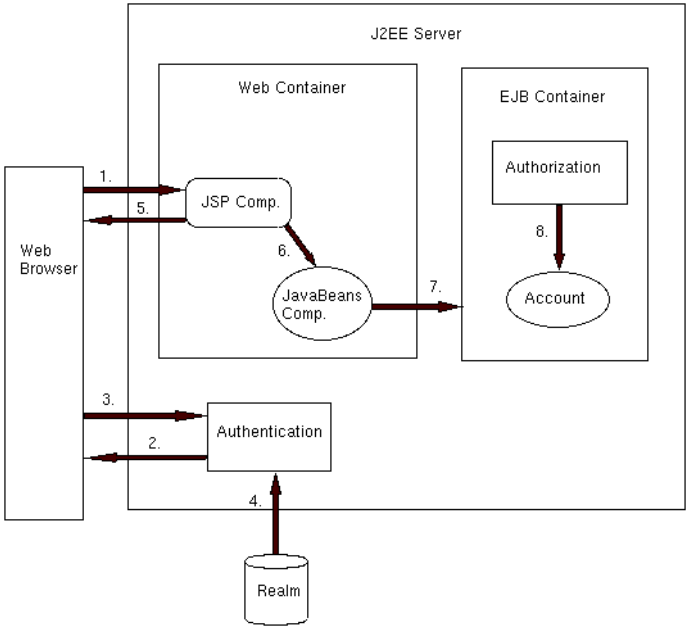

FIGURE 8-2 Authenticated Access to a JSP Component and an Enterprise Bean

Bean-Managed Security

The security mechanisms described in the Authentication and Authorization sections are sufficient for most J2EE applications. You control these mechanisms by declaring certain parameters with the Application Deployment Tool. Because this approach is declarative, you don't have to code your own security routines.

Some applications have special security requirements. For example, an application might make authorization decisions based on the time of day, the parameters of a call, or the internal state of an enterprise bean. Another application might restrict access based on user information stored in a database. If your application has special security requirements, you may want to take advantage of the APIs described in the following sections.

Getting the Caller's J2EE User

The `getCallerPrincipal` method of the `EJBContext` interface returns the `java.security.Principal` object that identifies the caller of the enterprise bean. (In this case, a principal is the same as a user.) In the following example, the `getUser` method of an enterprise bean returns the name of the J2EE user that invoked it:

```
public String getUser() {
   return context.getCallerPrincipal().getName();
}
. . .
public void setSessionContext(SessionContext context) {
   this.context = context;
}
```

To determine the caller of a servlet, you invoke the `getUserPrincipal` method.

Determining the Caller's Role

You can determine whether an enterprise bean's caller belongs to a particular role by invoking the `isCallerInRole` method:

```
boolean result = context.isCallerInRole("Customer");
```

You should declare the coded name (Customer) in the Security dialog box of the New Enterprise Bean wizard of the Application Deployment Tool. When you are ready to deploy the application, you must link the coded name with a role name. For example, to link the Customer coded name with the Buyer role name, you would follow these steps:

1. Select the Security tabbed pane of the enterprise bean.
2. If the Customer entry does not appear in the Coded Name column, click Add and enter Customer in that column.
3. If the Buyer role name is not listed in the Method Permissions table, click Edit Roles and add Buyer in the Editing Roles dialog box.
4. Go to the table at the top of the Security tabbed pane and locate the row that lists Customer in the Coded Name column. In that row, select Buyer from the Role Name combo box.

Because a coded name is linked to a role name, you may change the role name later on without having to change the coded name. For example, if you were to change the role name from Buyer to Shopper, you wouldn't have to change the Customer name in the code. However, you would have to relink the Customer coded name to the Shopper role name.

To determine the caller's role for a servlet, you invoke the `isUserInRole` method.

Security Policy Files

The J2EE server policy file is named `server.policy`. It resides in the `$J2EE_HOME/lib/security` directory. The J2EE application client policy file, `client.policy`, resides in the same directory.

For more information on security policy files, see the Security in JDK 1.2 chapter of the Java™ Tutorial at java.sun.com/docs/books/tutorial/security1.2/TOC.html.

Setting Up a Server Certificate

Certificates are used with the HTTPS protocol to authenticate Web browser clients. (For more information on certificates, see the Security in JDK 1.2 chapter of the Java™ Tutorial.) Unless a server certificate is installed, the HTTPS service of the J2EE server will not run. To set up a J2EE server certificate, follow these steps:

1. Generate a key pair and a self-signed certificate.

   The `keytool` utility enables you to create the certificate. The `keytool` that ships with the J2EE SDK has the same syntax as the one shipped with the Java 2™ Standard Edition. However, the J2EE SDK version programatically adds a Java™ Cryptographic Extension provider that has impementations of RSA algorithms. This provider enables you to import RSA signed certificates.

   To generate the certificate you run `keytool` as follows:

```
keytool -genkey -keyalg RSA -alias <certificate-alias>
```

   In the previous command, substitute `<certificate-alias>` with the alias of your certificate.

   The `keytool` utility prompts you for the following information:

   keystore password - The default value of this password is `changeit`. You may change the password by editing the `config/auth.properties` file.

   first and last name - Enter the fully-qualified name of your server. (This fully-qualified name includes the host name and the domain name.)

   organizational unit - Enter the appropriate value.

   organization - Enter the appropriate value.

   city or locality - Enter the appropriate value.

   state or province - Enter the unabbreviated name.

   two-letter country code - For the USA, the two-letter country code is US.

   key password for alias - Do not enter a password. Press Return.

2. Import the certificate.

   If your certificate will be signed by a Certification Authority (CA) other than Verisign, then you must import the CA certificate. Otherwise, you may skip this step. (Even if your certificate will be signed by verisign Test CA, you must import it.) To import the certificate, perform these tasks:

   a. Request the CA certificate from your CA. Store the certificate in a file.

b. To install the CA certificate in the Java 2 Standard Edition, run `keytool` as follows. (You must have the required permissions to modify the `$JAVA_HOME/jre/lib/security/cacerts` file.)

```
keytool -import -trustcacerts -alis <ca-cert-alias> -file <ca-cert-file-name>
```

3. Generate a Certificate Signing Request (CSR).

```
keytool -certreq -sigalg MD5withRSA -alias <cert-alias> -file <csr-filename>
```

4. Send the contents of the `<csr-filename>` for signing.

If you are using Verisign CA, go to http://digitalid.verisign.com/. Verisign will send the signed certificate in email. Store this certificate in a file.

5. Import the signed certificate that you recieved in email into the server.

```
keytool -import -alias <cert-alias> -file <signed-cert-file>
```

CHAPTER 9

Advanced Topics

If you've mastered the basic concepts of enterprise beans, you're ready for some advanced material:

- Mapping Table Relationships to Entity Beans
- Sending Email from an Enterprise Bean
- Connecting to a URL in an Enterprise Bean
- Accessing Enterprise Beans Through JSP Tag Libraries
- Deployment: Behind the Scenes

For information on designing full-scale J2EE applications, see the J2EE Blueprints web site (http://java.sun.com/j2ee/blueprints/index.html).

Mapping Table Relationships to Entity Beans

n a relational database, tables can be related by common columns. The relationships between the tables affect the design of their corresponding entity beans. The entity beans discussed in this section are backed up by tables with the following types of relationships:

- One-to-One Relationships
- One-to-Many Relationships
- Many-to-Many Relationships

Note: The following material applies only to entity beans with bean-managed persistence. In this release, container-managed persistence does not support table relationships.

One-to-One Relationships

In a *one-to-one* relationship, each row in a table is related to a single row in another table. For example, in a warehouse application a `storagebin` table might have a one-to-one relationship with a `widget` table. This application would model a physical warehouse where each storage bin contains one type of widget and each widget resides in one storage bin.

Figure 9-1 illustrates the `storagebin` and `widget` tables. Because the `storagebinid` uniquely identifies a row in the `storagebin` table, it is that table's primary key. The `widgetid` is the primary key of the `widget` table. The two tables are related because the `widgetid` is also a column in the `storagebin` table. By referring to the primary key of the `widget` table, the `widgetid` in the `storagebin` table identifies which widget resides in a particular storage bin in the warehouse. Because the `widgetid` of the `storagebin` table refers to the primary key of another table, it is called a *foreign key*. (The figure denotes a primary key with PK and a foreign key with FK.)

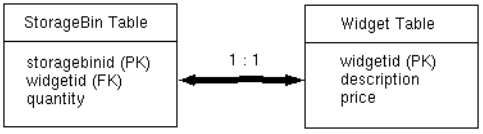

FIGURE 9-1 One-to-One Table Relationship

A dependent (child) table includes a foreign key that matches the primary key of the referenced (parent) table. The values of the foreign keys in the `storagebin` (child) table depend on the primary keys in the `widget` (parent) table. For example, if the `storagebin` table has a row with a `widgetid` of 344, then the `widget` table should also have a row whose `widgetid` is 344.

When designing a database application, you may choose to enforce the dependency between the parent and child tables. There are two ways to enforce such a dependency: by defining a referential constraint in the database or by performing checks in the application code. The `storagebin` table has a referential constraint named `fk_widgetid`:

```
create table storagebin
  (storagebinid varchar(3)
     constraint pk_storagebin primary key,
  widgetid varchar(3),
  quantity integer,
     constraint fk_widgetid
     foreign key (widgetid)
```

```
    references widget(widgetid));
```

The StorageBinEJB and WidgetEJB classes illustrate the one-to-one relationship of the storagebin and widget tables. (The source code for the classes is in the doc/guides/ejb/examples/storagebin directory.)

The StorageEJB class contains variables for each column in the storagebin table, including the foreign key, widgetId:

```
private String storageBinId;
private String widgetId;
private int quantity;
```

The ejbFindByWidgetId method of the StorageEJB class returns the storageBinId that matches a given widgetId:

```
public String ejbFindByWidgetId(String widgetId)
   throws FinderException {

   String storageBinId;

   try {
      storageBinId = selectByWidgetId(widgetId);
    } catch (Exception ex) {
        throw new EJBException("ejbFindByWidgetId: " +
           ex.getMessage());
    }

   if (storageBinId == null) {
      throw new ObjectNotFoundException
         ("Row for widgetId " + widgetId + " not found.");
   }
   else {
      return storageBinId;
   }
}
```

The ejbFindByWidgetId method locates the widgetId by querying the database in the selectByWidgetId method:

```
private String selectByWidgetId(String widgetId)
   throws SQLException {
```

```
    String storageBinId;

    String selectStatement =
          “select storagebinid “ +
          “from storagebin where widgetid = ? “;
    PreparedStatement prepStmt =
          con.prepareStatement(selectStatement);
    prepStmt.setString(1, widgetId);

    ResultSet rs = prepStmt.executeQuery();

    if (rs.next()) {
       storageBinId = rs.getString(1);
    }
    else {
       storageBinId = null;
    }

    prepStmt.close();
    return storageBinId;
}
```

To find out which storage bin a widget resides in, the `StorageBinClient` program calls the `findByWidgetId` method:

```
String widgetId = “777”;
StorageBin storageBin = storageBinHome.findByWidgetId(widgetId);
String storageBinId = (String)storageBin.getPrimaryKey();
int quantity = storageBin.getQuantity();
```

Tips for running the `StorageBinEJB` example:

- Create the database tables by running the `cloudTable.sh` (UNIX) or `cloudTable.bat` (Windows) script from the command line prompt. For example, you should run the `cloudTable.bat` script as follows:

```
cd %J2EE_HOME%\doc\guides\ejb\examples\storagebin
..\util\cloudTable.bat
```

- Specify bean-managed persistence for both entity beans.
- For the business methods, specify the Required transaction attribute.

- For the `StorageBinBean`, specify `jdbc/StorageBinDB` as the coded name for the resource reference.
- For the `WidgetBean`, specify `jdbc/WidgetDB` as the coded name for the resource reference.
- Use the JNDI names listed in the following table.

TABLE 9-1 JNDI Names for the StorageBinEJB Example

Component/Reference Name	JNDI Name
StorageBinBean	MyStorageBin
jdbc/StorageBinDB	jdbc/Cloudscape
WidgetBean	MyWidget
jdbc/WidgetDB	jdbc/Cloudscape

One-to-Many Relationships

If the primary key in a parent table matches multiple foreign keys in a child table, then the relationship is *one-to-many*. This relationship is common in database applications. For example, an application for a sports league might access a `team` table and a `player` table. Each team has multiple players and each player belongs to a single team. Every row in the child table (`player`), has a foreign key identifying the player's team. This foreign key matches the `team` table's primary key.

The sections that follow describe how you might implement one-to-many relationships in entity beans. When designing such entity beans, you must decide whether both tables are represented by entity beans, or just one.

A Helper Class for the Child Table

Not every database table needs to be mapped to an entity bean. If a database table doesn't represent a business entity, or if it stores information that is contained in another entity, then the table should be represented with a helper class. For example, in an online shopping application each order submitted by a customer can have multiple line items. The application stores the information in the database tables shown by the following figure.

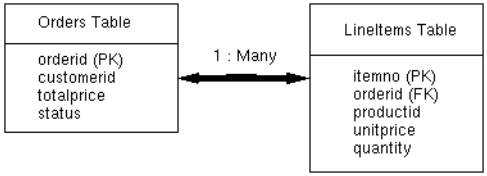

FIGURE 9-2 One-to-Many Relationship: Order and Line Items

Not only does a line item belong to an order, it does not exist without the order. Therefore, the `lineitems` table should be represented with a helper class and not with an entity bean. Using a helper class in this case is not required, but doing so might improve performance because a helper class uses fewer system resources than an entity bean.

The `LineItem` and `OrderEJB` classes show how to implement a one-to-many relationship with a helper class (`LineItem`) and an entity bean (`OrderEJB`). (The source code for the classes is in the `doc/guides/ejb/examples/order` directory.) The instance variables in the `LineItem` class correspond to the columns in the `lineitems` table. The `itemNo` variable matches the primary key for the `lineitems` table and the `orderId` variable represents the table's foreign key. Here is the source code for the `LineItem` class:

```
public class LineItem implements java.io.Serializable {

    String productId;
    int quantity;
    double unitPrice;
    int itemNo;
    String orderId;


    public LineItem(String productId, int quantity,
      double unitPrice, int itemNo, String orderId) {

       this.productId = productId;
       this.quantity = quantity;
       this.unitPrice = unitPrice;
       this.itemNo = itemNo;
       this.orderId = orderId;
    }
```

```
    public String getProductId() {
        return productId;
    }

    public int getQuantity() {
        return quantity;
    }

    public double getUnitPrice() {
        return unitPrice;
    }

    public int getItemNo() {
        return itemNo;
    }

    public String getOrderId() {
        return orderId;
    }

}
```

The `OrderEJB` class contains an `ArrayList` variable named `lineItems`. Each element in the `lineItems` variable is a `LineItem` object. The `lineItems` variable is passed to the `OrderEJB` class in the `ejbCreate` method. For every `LineItem` object in the `lineItems` variable, the `ejbCreate` method inserts a row into the `lineitems` table. It also inserts a single row into the `orders` table. The code for the `ejbCreate` method follows:

```
public String ejbCreate(String orderId, String customerId,
    String status, double totalPrice, ArrayList lineItems)
    throws CreateException {

    try {
        insertOrder(orderId, customerId, status, totalPrice);
        for (int i = 0; i < lineItems.size(); i++) {
            LineItem item = (LineItem)lineItems.get(i);
            insertItem(item);
```

```
        }
    } catch (Exception ex) {
        throw new EJBException("ejbCreate: " +
            ex.getMessage());
    }

    this.orderId = orderId;
    this.customerId = customerId;
    this.status = status;
    this.totalPrice = totalPrice;
    this.lineItems = lineItems ;

    return orderId;
}
```

The `OrderClient` program creates and loads an `ArrayList` of `LineItem` objects. The program passes this `ArrayList` to the entity bean when it invokes the `create` method:

```
ArrayList lineItems = new ArrayList();
lineItems.add(new LineItem("p23", 13, 12.00, 1, "123"));
lineItems.add(new LineItem("p67", 47, 89.00, 2, "123"));
lineItems.add(new LineItem("p11", 28, 41.00, 3, "123"));
. . .
Order duke = home.create("123", "c44", "open",
   totalItems(lineItems), lineItems);
```

Other methods in the `OrderEJB` class also access both database tables. The `ejbRemove` method, for example, deletes not only a row from the `orders` table, but also deletes all corresponding rows in the `lineitems` table. The `ejbLoad` and `ejbStore` methods synchronize the state of an `OrderEJB` instance, including the `lineItems ArrayList`, with the `orders` and `lineitems` tables.

The `ejbFindByProductId` method enables clients to locate all orders that have a particular line item. This method queries the `lineitems` table for all rows with a particular `productId`. The method returns a `Collection` of `productId String` objects. The `OrderClient` program iterates through the `Collection` and prints the primary key of each order:

```
Collection c = home.findByProductId("p67");
Iterator i=c.iterator();
```

```
while (i.hasNext()) {
   Order order = (Order)i.next();
   String id = (String)order.getPrimaryKey();
   System.out.println(id);
 }
```

Tips for running the `OrderEJB` example:

- Add the `LineItem.class` file to the EJB .jar file that contains the `OrderEJB.class` file.
- Specify `jdbc/OrderDB` as the coded name for the resource reference.
- For the transaction attributes of the business methods, specify Required. This attribute value causes the container to call `ejbLoad` before (and `ejbStore` after) each business method invocation. These calls will synchronize the bean's state with the database tables.
- Use the JNDI names listed in the following table.

TABLE 9-2 JNDI Names for the OrderEJB Example

Component/Reference Name	JNDI Name
OrderBean	MyOrder
jdbc/OrderDB	jdbc/Cloudscape

An Entity Bean for the Child Table

You should consider building an entity bean for a child table under the following conditions:

- The information in the child table is not a subset of that in the parent table.
- The business entity of the child table could exist without that of the parent table.
- The child table might be accessed by another application that does not access the parent table.

These conditions exist in the following scenario. Suppose that each sales representative in a company has multiple customers and that each customer has only one sales representative. The company tracks its sales force with a database application. In the database, each row in the `salesrep` table (parent) matches multiple rows in the `customer` table (child). Figure 9-4 illustrates this relationship.

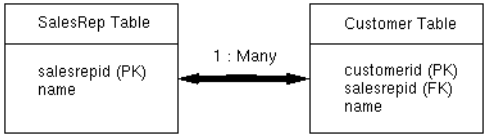

FIGURE 9-3 One-to-Many Relationship: Sales Representative and Customers

The `SalesRepEJB` and `CustomerEJB` entity bean classes implement the one-to-many relationship of the of `sales` and `customer` tables. (The source code for these classes is in the `doc/guides/ejb/examples/salesrep` directory.)

The `SalesRepEJB` class contains a variable named `customerIds`, which is an `ArrayList` of `String` elements. These `String` elements identify which customers belong to the sales representative. Because the `customerIds` variable reflects this relationship, the `SalesRepEJB` class must keep the variable up to date.

The `SalesRepEJB` class instantiates the `customerIds` variable in the `setEntityContext` method, not in `ejbCreate`. The container invokes `setEntityContext` just once-- when it creates the bean instance-- ensuring that `customerIds` is instantiated just once. Because the same bean instance can assume different identities during its life cycle, instantiating `customerIds` in `ejbCreate` might cause multiple and unnecessary instantiations. Therefore, the `SalesRepEJB` class instantiates the `customerIds` variable in `setEntityContext`:

```
public void setEntityContext(EntityContext context) {

   this.context = context;
   customerIds = new ArrayList();

   try {
      makeConnection();
      Context initial = new InitialContext();
      Object objref = initial.lookup("java:comp/env/ejb/Customer");

      customerHome =
        (CustomerHome)PortableRemoteObject.narrow(objref,
         CustomerHome.class);
   } catch (Exception ex) {
       throw new EJBException("setEntityContext: " +
          ex.getMessage());
   }
```

```
}
```

Invoked by the ejbLoad method, loadEnrollerIds is a private method that refreshes the customerIds variable. There are two approaches when coding a method such as loadCustomerIds: fetch the identifiers from the customer database table or get them from the Customer entity bean. Fetching the identifiers from the database might be faster, but exposes the SalesRepEJB code to the Customer bean's underlying database table. In the future, if you were to change the Customer bean's table (or move the bean to a different J2EE server), then you might need to change the SalesRepEJB code. But if the SalesRepEJB gets the identifiers from the Customer entity bean, no coding changes would be required. The two approaches present a trade-off: performance versus flexibility. The SalesRepEJB example opts for flexibility, loading the customerIds variable by calling the findSalesRep and getPrimaryKey methods of the Customer bean. Here is the code for the loadCustomerIds method:

```
private void loadCustomerIds() {

   customerIds.clear();

   try {
      Collection c = customerHome.findBySalesRep(salesRepId);
      Iterator i=c.iterator();

      while (i.hasNext()) {
         Customer customer = (Customer)i.next();
         String id = (String)customer.getPrimaryKey();
         customerIds.add(id);
      }

   } catch (Exception ex) {
       throw new EJBException("Exception in loadCustomerIds: " +
          ex.getMessage());
    }
}
```

If a customer's sales representative changes, the client program updates the database by calling the setSalesRepId method of the CustomerEJB class. The next time a business method of the SalesRepEJB is called, the ejbLoad method invokes loadCustomerIds, which refreshes the customerIds variable. (To ensure that ejbLoad is invoked before each business method, set the transaction attributes of the business methods to Required.) For example, the SalesRepClient program changes the salesRepId for a customer named Mary Jackson:

```
Customer mary = customerHome.findByPrimaryKey(“987”);
mary.setSalesRepId(“543”);
```

The `salesRepId` 543 identifies a sales representative named Janice Martin. To list all of Janice’s customers, the `SalesRepClient` program invokes the `getCustomerIds` method, iterates through the `ArrayList` of identifiers, and locates each `Customer` bean by calling the `Customer` bean’s `findByPrimaryKey` method:

```
SalesRep janice = salesHome.findByPrimaryKey("543");
ArrayList a = janice.getCustomerIds();
i = a.iterator();

while (i.hasNext()) {
   String customerId = (String)i.next();
   Customer customer = customerHome.findByPrimaryKey(customerId);
   String name = customer.getName();
   System.out.println(customerId + ": " + name);
}
```

Tips for running the `SalesRepEJB` example:

- For both beans specify `jdbc/SalesDB` as the coded name for the resource reference.
- For the transaction attributes of the business methods, specify Required.
- Because the `SalesRepEJB` class refers to the `Customer` bean, you must specify the EJB reference using the values in the following table.

TABLE 9-3 EJB Reference in SalesRepEJB

Dialog Field	Value
Coded Name	ejb/Customer
Type	Entity
Home	CustomerHome
Remote	Customer

- Use the JNDI names listed in the following table.

TABLE 9-4 JNDI Names for the SalesRepEJB Example

Component/Reference Name	JNDI Name
SalesRepBean	MySalesRep
CustomerBean	MyCustomer
jdbc/SalesDB	jdbc/Cloudscape
ejb/Customer	MyCustomer

Many-to-Many Relationships

In a *many-to-many* relationship, each entity may be related to multiple occurrences of the other entity. For example, a college course has many students and each student may take several courses. In a database, this relationship is represented by a cross reference table containing the foreign keys. In figure 9-5, the cross reference table is the `enrollment` table. (PK indicates a primary key and FK a foreign key.)

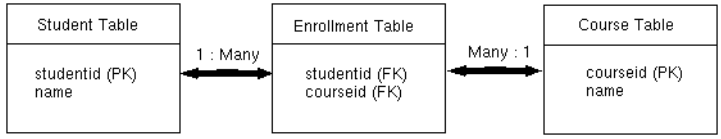

FIGURE 9-4 Many-to-Many Relationship: Students and Courses

These tables are accessed by the `StudentEJB`, `CourseEJB`, and `EnrollerEJB` classes. (The sample code for these classes is in the `doc/guides/ejb/examples/enroller` directory.)

The `StudentEJB` and `CourseEJB` classes are complementary. Each class contains an `ArrayList` of foreign keys. The `StudentEJB` class, for example, contains an `ArrayList` named `courseIds`, which identifies the courses the student is enrolled in. The `ejbLoad` method adds elements to the `courseIds` `ArrayList` by calling `loadCourseIds`, a private method. The `loadCourseIds` method gets the course identifiers from the `Enroller` session bean. The source code for the `loadCourseIds` method follows:

```
private void loadCourseIds() {

   courseIds.clear();
```

```
   try {
      Enroller enroller = enrollerHome.create();
      ArrayList a = enroller.getCourseIds(studentId);
      courseIds.addAll(a);

   } catch (Exception ex) {
       throw new EJBException("Exception in loadCourseIds: " +
          ex.getMessage());
    }
}
```

Invoked by the `loadCourseIds` method, the `getCourses` method of the `EnrollerEJB` class queries the `enrollment` table:

```
select courseid from enrollment
where studentid = ?
```

Only the `EnrollerEJB` class accesses the `enrollment` table. Therefore, the `EnrollerEJB` class manages the student-course relationship represented in the `enrollment` table. If a student enrolls in a course, for example, the client calls the `enroll` business method, which inserts a row:

```
insert into enrollment
values (studentid, courseid)
```

If a student drops a course, the `unEnroll` method deletes a row:

```
delete from enrollment
where studentid = ? and courseid = ?
```

And if a student leaves the school, the `deleteStudent` method deletes all rows in the table for that student:

```
delete from enrollment
where student = ?
```

The `EnrollerEJB` class does not delete the matching row from the `student` table. That action is performed by the `ejbRemove` method of the `StudentEJB` class. To ensure that both deletes are executed as a single operation, they should belong to the same transaction. (See the Transactions chapter for more information.)

Tips for running the `EnrollerEJB` example:

- For all three beans, specify `jdbc/CollegeDB` as the coded name for the resource reference.
- For all three beans, specify container-managed transactions.

- For the transaction attributes of the business methods, specify `Required`.
- For the `Student` and `Course` entity beans, you must specify an EJB reference to the `Enroller` session bean using the values in the following table.

TABLE 9-5 EJB Reference in StudentEJB and CourseEJB

Dialog Field	Value
Coded Name	ejb/Enroller
Type	Session
Home	EnrollerHome
Remote	Enroller

- Use the JNDI names listed in the following table.

TABLE 9-6 JNDI Names for the EnrollerEJB Example

Component/Reference Name	JNDI Name
EnrollerBean	MyEnroller
StudentBean	MyStudent
CourseBean	MyCourse
jdbc/CollegeDB	jdbc/Cloudscape
ejb/Enroller	MyEnroller

Sending Email from an Enterprise Bean

If you've ever ordered a product from a web site, you've probably received an email confirming your order. The `ConfirmerEJB` class demonstrates how to send email from an enterprise bean. (The sample code is in the `doc/guides/ejb/examples/confirmer` directory.)

In the sendNotice method of the ConfirmerEJB class, the lookup method returns a Session object, which represents a mail session. Like a database connection, a mail session is a resource. As with any resource, you must link the coded name (TheMailSession) with a JNDI name in the Resource References dialog box of the New Enterprise Bean Wizard. (See table 9-7). Using the Session object as an argument, the the sendNotice method creates an empty Message object. After calling several set methods on the Message object, sendNotice invokes the send method of the Transport class to send the message on its way. The source code for the sendNotice method follows:

```
public void sendNotice(String recipient) {

   try {
       Context initial = new InitialContext();
       Session session =
         (Session) initial.lookup(“java:comp/env/TheMailSession”);

       Message msg = new MimeMessage(session);
       msg.setFrom();

       msg.setRecipients(Message.RecipientType.TO,
          InternetAddress.parse(recipient, false));

       msg.setSubject(“Test Message from ConfirmerEJB”);

       DateFormat dateFormatter = DateFormat.getDateTimeInstance(
          DateFormat.LONG, DateFormat.SHORT);

       Date timeStamp = new Date();

       String messageText = “Thank you for your order.” + ‘\n’ +
          “We received your order on “ +
          dateFormatter.format(timeStamp) + “.”;

       msg.setText(messageText);
       msg.setHeader(“X-Mailer”, mailer);
       msg.setSentDate(timeStamp);

       Transport.send(msg);
```

```
    } catch(Exception e) {
        throw new EJBException(e.getMessage());
    }
}
```

Tips for running the `ConfirmerEJB` example:

- Before compiling the `ConfirmerClient` program, change the value of the `recipient` variable to an actual email address.
- In the Resource References dialog box of the New Enterprise Bean wizard, specify the resource reference for the mail session with the values in the following table.

TABLE 9-7 Resource Reference Dialog for the `ConfirmerEJB` Example

Dialog Field	Value
Coded Name	TheMailSession
Type	javax.mail.Session
Authentication	Application

- In the Resource References tab of the enterprise bean, specify the values in the following table.

TABLE 9-8 Resource Reference Tab for the `ConfirmerEJB` Example

Field	Value
JNDI Name	MyMailer
From	(your email address)
Host	(mail server host)
User Name	(your UNIX or Windows user name)

- Use the JNDI names listed in the following table.

TABLE 9-9 JNDI Names for the `ConfirmerEJB` Example

Component/Reference Name	JNDI Name
ConfirmerBean	MyConfirmer
TheMailSession	MyMailer

Note: If the application cannot connect to the mail server it will generate this exception:

```
javax.mail.MessagingException: Could not connect to SMTP host
```

To fix this problem, make sure that the mail server is running and that you've entered the correct name for the mail server host in the Resource Reference tab.

Connecting to a URL in an Enterprise Bean

A Uniform Resource Locator (URL) specifies the location of a resource on the Web. The `HTMLReaderEJB` class shows how to connect to a URL from within enterprise bean. (The sample code is in the `doc/guides/ejb/examples/urlconnect` directory.)

The `getContents` method of the `HTMLReaderEJB` class returns a `String` that contains the contents of an HTML file. This method method looks up the `java.net.URL` object associated with a coded name (`url/MyURL`), opens a connection to it, and then reads its contents from an `InputStream`. Before deploying the application, you must map the coded name (`url/MyURL`) to a JNDI name (a URL string). Here is the source code for the `getContents` method:

```
public StringBuffer getContents() throws HTTPResponseException {

    Context context;
    URL url;
    StringBuffer buffer;
    String line;
    int responseCode;
    HttpURLConnection connection;
```

```
    InputStream input;
    DataInputStream dataInput;

    try {
       context = new InitialContext();
       url = (URL)context.lookup("java:comp/env/url/MyURL");
       connection = (HttpURLConnection)url.openConnection();
       responseCode = connection.getResponseCode();
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }

    if (responseCode != HttpURLConnection.HTTP_OK) {
       throw new HTTPResponseException("HTTP response code: " +
          String.valueOf(responseCode));
    }

    try {
       buffer = new StringBuffer();
       input = connection.getInputStream();
       dataInput = new DataInputStream(input);
       while ((line = dataInput.readLine()) != null) {
          buffer.append(line);
          buffer.append('\n');
       }
    } catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }

    return buffer;
}
```

Tips for running the `HTMLReaderEJB` example:

- Include the `HTTPResponseException` class in the enterprise bean.
- In the Resource References dialog box of the New Enterprise Bean wizard, specify the values in the following table. (Replace the `<host>` string with the name of the host running the J2EE server.)

TABLE 9-10 Resource References Dialog for the `HTMLReaderEJB` Example

Dialog Field	Value
Coded Name	url/MyURL
Type	java.net.URL
Authentication	Container
URL	http://<host>:8000/index.html

- Use the JNDI names listed in the following table. The JNDI name for the `url/MyURL` entry should match the URL field of the Resource References dialog box.

TABLE 9-11 JNDI Names for the `HTMLReaderEJB` Example

Component/Reference Name	JNDI Name
HTMLReaderBean	MyHTMLReader
url/MyURL	http://<host>:8000/index.html

The URL specified in the preceding tables refers to the the default `public_html/index.html` file of your J2EE installation. To change this URL, go to the Resource Refs tabbed pane for the enterprise bean, select the entry in the table, and edit the URL field.

To connect to a URL outside of your firewall, you must perform these tasks:

1. Exit the Application Deployment Tool.

2. Stop the J2EE server.

3. In the `bin/j2ee` script, add the following options to the PROPS environment variable:

```
-Dhttp.proxyPort=<port> -Dhttp.proxyHost=<host>
```

The `<port>` is the proxy's port number and `<host>` is the name of your proxy host.

4. In the `lib/security/Server.policy` file, edit the following line:

```
permission java.net.SocketPermission "*:0-65535", "connect";
```

Modify the line so that it appears as follows::

```
permission java.net.SocketPermission "*", "connect";
```

5. Start the J2EE server.

6. Start the Application Deployment Tool.

Accessing Enterprise Beans Through JSP Tag Libraries

A tag library enables you to define new actions for a JSP page. For example, you can define an action that locates the home interface of an enterprise bean. The sections that follow show how to create such a tag library in a J2EE application named `ConverterJSPApp`. This application contains the following elements:

- JSP File: `Converter.jsp`
- Tag Library Descriptor File: `taglib.tld`
- Tag Handler Class: `EjbTag.java`
- TagExtraInfo Class: `EjbExtraInfo`
- Session Bean: `ConverterEJB`

The `ConverterEJB.java` file is in the `doc/guides/ejb/examples/converter` directory. The other files are in the `doc/guides/ejb/examples/jsptag` directory.

Setting Up the ConverterJSPApp Application

Before creating the web component for the JSP page, you should perform these tasks:

- Open the `ConverterApp` you created in the Getting Started chapter.
- In the `ConverterApp`, save the `ConverterBean` component in a file named `ConverterEJB.jar`.
- Create a J2EE application named `ConverterJSPApp`.
- Add the `ConverterEJB.jar` file to the `ConverterJSPApp` application.

Writing the JSP File

This section briefly describes the JSP tags in the `Converter.jsp` file. These tags are marked by bold font in the full listing of `Converter.jsp` included at the end of this section. (For more information on writing JSP files, see the JavaServer Pages web site at java.sun.com / products / jsp / index.html.)

The `taglib` directive is the first JSP tag in the `Converter.jsp` file. This directive identifies the tag library descriptor (`taglib.tld`) and defines the tag prefix (`j2ee`) that associates subsequent tags with the tag library:

```
<%@ taglib uri=”taglib.tld” prefix=”j2ee” %>
```

The tag library described in the `taglib.tld` file has just one tag, which is named `ejb`. The `ejb` tag has three attributes: `jndiName`, `homeInterface`, and `homeVar`. The `ejb` tag file assigns values to each attribute:

```
<j2ee:ejb
   jndiName="java:comp/env/ejb/MyConverter"
   homeInterface="ConverterHome"
   homeVar="converterHome">
   <% converter = converterHome.create(); %>
</j2ee:ejb>
```

The `Converter.jsp` file uses script elements in conjunction with the tag library. The first of these elements declares the `Converter` session bean:

```
<% Converter converter = null; %>
```

The next script element creates a new session bean:

```
<% converter = converterHome.create(); %>
```

The last two script elements invoke methods on the session bean, returning values which are displayed:

```
<%= converter.dollarToYen(100.00) %>
<%= converter.yenToEuro(100.00) %>
```

The full listing for the Converter.jsp file follows:

```
<html>
<%@ taglib uri=”taglib.tld” prefix=”j2ee” %>

<head>
    <title>Converter JSP</title>
</head>

<h1><b><center>Converter JSP Example</center></b></h1>
<hr>


<% Converter converter = null; %>
```

```
<j2ee:ejb
   jndiName="java:comp/env/ejb/MyConverter"
   homeInterface="ConverterHome"
   homeVar="converterHome">
   <% converter = converterHome.create(); %>
</j2ee:ejb>


<p>
dollarToYen:  <%= converter.dollarToYen(100.00) %>
<p>
yenToEuro: <%= converter.yenToEuro(100.00) %>

</html>
```

Writing the Tag Library Descriptor

A tag library descriptor is an XML file whose elements describe a particular tag library. A JSP container uses the tag library descriptor to interpret pages that include `taglib` directives. The `taglib` directive in the `Converter.jsp` page refers to the `taglib.tld` tag library descriptor:

```
<%@ taglib uri="taglib.tld" prefix="j2ee" %>
```

A listing of the `taglib.tld` file follows. The `tag` element defines the `ejb` action, including its three attributes (`jndiName`, `homeInterface`, and `homeVar`). The `tagclass` element defines the tag handler class (`EjbTag`). The `teiclass` element specifies the `TagExtraInfo` class (`EjbExtraInfo`).

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
      PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
      "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tab library descriptor -->

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <urn></urn>
```

```
  <info>
    Tag library for EJB support
  </info>

  <!-- ejb tag -->

  <tag>
    <name>ejb</name>
    <tagclass>EjbTag</tagclass>
    <teiclass>EjbExtraInfo</teiclass>
    <bodycontent>JSP</bodycontent>
    <info>
       Look up home interface and declare enterprise bean.
    </info>

    <attribute>
      <name>jndiName</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
      <type>String</type>
    </attribute>
    <attribute>
      <name>homeInterface</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>homeVar</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>
```

Coding the Tag Handler Class

A tag handler is an object in the web container that helps evaluate actions when a JSP page executes. The `EjbTag` class, for example, is the tag handler for the `ejb` action. The `doInitBody` method of the `EjbTag` class sets the `homeVar` attribute to the reference returned by a JNDI `lookup` method:

```
public void doInitBody() throws JspException {
    try {
        System.out.println("doInitBody()");
        InitialContext ic = new InitialContext();
        Object homeRef = ic.lookup(jndiName);
        homeRef = PortableRemoteObject.narrow(homeRef,
           Class.forName(homeInterface));
        pageContext.setAttribute(homeVar, homeRef);
    } catch (NamingException ex) {
       throw new JspTagException("Unable to lookup home: "+jndiName);
    } catch (ClassNotFoundException ex) {
      throw new JspTagException("Class "+homeInterface+" not found");
    }
}
```

Coding the TagExtraInfo Class

If a tag defines scripting variables or if it validates attributes during translation, then you must provide a `TagExtraInfo` class. A subclass of `TagExtraInfo`, the `EjbExtraInfo` class implements the `getVariableInfo` method. This method provides information about the `homeVar` and `homeInterface` attributes. The `EjbExtraInfo` class follows:

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class EjbExtraInfo extends TagExtraInfo {
   public VariableInfo[] getVariableInfo(TagData data) {
      return new VariableInfo[]
             {
                 new VariableInfo(
                    (String)data.getAttribute("homeVar"),
```

```
                    (String)data.getAttribute("homeInterface"),
                    true,
                    VariableInfo.NESTED),
            };
    }
```

To compile the `EjbTag` and `TagExtraInfo` classes, change to the `examples/jsptag` directory and execute these commands:

UNIX:

```
CPATH=.:$J2EE_HOME/lib/j2ee.jar:ejb.jar
javac -classpath “$CPATH” *.java
```

Windows:

```
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar;ejb.jar
javac -classpath %CPATH% *.java
```

Creating the Tag Library’s .war File

To create a .war file, you run the New Web Component Wizard of the Application Deployment Tool. To start the wizard, from the File menu choose New Web Component. The wizard displays the following dialog boxes. (You may skip any dialog boxes not listed here.)

WAR File General Properties Dialog Box:

a. In the combo box labelled “Web Component Will Go In,” select ConverterJSPApp.

b. In the WAR Display Name field, enter `ConverterWAR`.

c. Click Add.

d. In the Add Content Files dialog box, choose the `examples/jsptag` directory. You may either type the directory name in the Root Directory field or locate it by clicking Browse.

c. Select the `Converter.jsp` and `taglib.tld` files from the text area and click Add.

d. Click Next.

e. Choose the `examples/jsptag` directory again.

f. Select the `EjbExtraInfo.class` and `EjbTag.class` files from the text area and click Add.

g. Click Finish.

h. Click Next.

Choose Component Type Dialog Box:

a. Select JSP.

b. Click Next.

Component General Properties Dialog Box:

a. In the JSP Filename combo box, select Converter.jsp.

b. In the Web Component Display Name field, enter `TheConverter`.

c. Click Next.

Enterprise Bean References Dialog Box:

a. Click Add.

b. In the Coded Name column enter `ejb/MyConverter`.

c. In the Type column select `Session`.

d. In the Home column enter `ConverterHome`.

e. In the Remote column enter `Converter`.

f. Click Finish.

Specifying the Web Context Root

1. In the tree view select ConverterJSPApp.

2. In the Web Context tabbed pane, enter `ConverterContextRoot` in the ContextRoot column.

Specifying the JNDI Names

In the JNDI Names tabbed pane for the `ConverterJSPApp`, specify `MyConverter` as the JNDI name for both the `ejb/Converter` reference and the `ConverterBean` component.

Deploying the ConverterJSPApp Application

1. From the Tools menu, choose Deploy Application.

2. In the first dialog box, do not select the checkbox labelled “Return Client Jar.”

3. In the second dialog box, verify the JNDI names.

4. In the third dialog box, verify the context root.

Running the ConverterJSPApp Application

To run `Converter.jsp` from your browser, specify the URL as follows, but replace `<host>` with the name of the machine that is running the J2EE server:

```
http://<host>:8000/ConverterContextRoot/Converter.jsp
```

Comparing the ConverterJSPApp and AccountJSPApp Applications

In the Clients chapter, the JavaServer Pages™ Components section describes a J2EE application named `AccountJSPApp`. Unlike the `ConverterJSPApp`, the `AccountJSPApp` accesses an enterprise bean through a JavaBeans™ component. This component mirrors the enterprise bean-- its state matches the enterprise bean’s state. The `ConverterJSPApp` does not have to synchronize the state of an intermediary object with that of the enterprise bean. However, the JSP page of the `ConverterJSPApp` cannot maintain a state throughout a session. Every time a web client invokes the `Converter.jsp` file, all of the page’s tags are executed. Because of this stateless nature, the approach taken by the `ConverterJSPApp` is not appropriate for an application that carries on a “conversation” with the end-user.

Deployment: Behind the Scenes

If you're an advanced user, you may want to know what happens inside the J2EE server when you deploy an application. Here's the story:

1. The server transfers the application .jar file from the `deploytool` process to the `j2ee` process.

After this step, the server uses the transferred copy of the application .jar file to do its work, and the .jar file that was manipulated by the deployment tool is left unchanged.

2. The `j2ee` process saves the application .jar file in its repository at this directory:

`$J2EE_HOME/repository/<hostname>/applications`

3. The `j2ee` process opens the application .jar file, reads the deployment descriptors, and for each bean generates the source code for the home interface and `EJBObject` implementation. These source code files are placed in the following directory:

`$J2EE_HOME/repository/<hostname>/gnrtrTMP`

4. The `j2ee` process compiles the home interface and the `EJBObject` implementations and then runs the `rmic` command on the class files.

This step creates the stubs and skeletons for the home and remote objects. The server sends the client .jar file to the deployer and saves the file with the name chosen at the start of the deployment process.

5. The server packages the generated classes into a server .jar file and stores the .jar file in the repository.

6. The server creates a client .jar file that contains the home and remote interfaces and the stubs for the home and remote objects.

The server sends the client .jar file to the deployer and saves the file according to the name chosen at the start of the deployment process.

The location of the client .jar file is added to the CLASSPATH environment variable on any client that calls the application. Then, at runtime, the appropriate stub classes can be loaded so that the client can successfully locate objects, for example, the home object for an enterprise bean in the application.

If the .jar file has any Web components, the EJB server copies the components to and installs them on the Web server.

7. If you started `j2ee` with the `-singleVM` option (the default), the `j2ee` process hosts the enterprise beans in its process and creates containers for them.

If you started the `j2ee` process with the `-multiVM` option, the server starts a process which loads the server .jar file and creates containers for the enterprise beans.

8. At this point, the deployment process is complete.

CHAPTER 10

Running the J2EE Tools

The J2EE SDK includes the following tools:

- Application Deployment Tool
- Cleanup Script
- Cloudscape Server
- J2EE Server
- Key Tool
- Packager
- Realm Tool
- Runclient Script
- Verifier

Application Deployment Tool

The Application Deployment Tool enables you to build and deploy J2EE components and applications. If you run the `deploytool` script with no options, the GUI version is launched:

```
deploytool
```

The GUI version includes online help information that is context sensitive. To access a help topic for a particular dialog box or tabbed pane, press f1. For a quick introduction to the tool, see the Getting Started chapter of this manual.

The command-line version of the tool also enables you to deploy and undeploy applications. The following table describes the tool's command-line options:

TABLE 10-1 Command-Line Options for deploytool Script

Option	Description
-deploy <myAppljar> <myServerName> [<myAppClientCode.jar>]	Deploys the J2EE application contained in the .ear file specified by <myAppljar> onto the J2EE server running on the machine specified by <myServerName>. Optionally, a .jar file for a stand-alone Java™ application client may be created by specifying <myAppClientCode.jar>.
-listApps <server-name>	Lists the J2EE applications that are deployed on the J2EE server running on the machine specified by <server-name>.
-uninstall <app-name> <server-name>	Undeploys the J2EE application whose name is <app-name> from the J2EE server running on the machine specified by <server-name>.
-help	Displays options.
-ui	Runs GUI version (default).

Cleanup Script

The `cleanup` script removes all deployed applications from your J2EE server. It will not delete the component files (.jar, .war, .ear). You run the script from the command line:

```
cleanup
```

Warning: Use this utility with care!

Cloudscape Server

The enterprise code examples in this manual have been tested with the Cloudscape DBMS, which is included in the J2EE SDK.

Starting and Stopping Cloudscape

Before your enterprise beans can access a Cloudscape database, you must run the Cloudscape server from the command line:

```
cloudscape -start
```

You should see output similar to the following:

```
Mon Aug 09 11:50:30 PDT 1999: [RmiJdbc] COM.cloudscape.core.JDBCDriver registered in
DriverManager
Mon Aug 09 11:50:30 PDT 1999: [RmiJdbc] Binding RmiJdbcServer...
Mon Aug 09 11:50:30 PDT 1999: [RmiJdbc] No installation of RMI Security Manager...
Mon Aug 09 11:50:31 PDT 1999: [RmiJdbc] RmiJdbcServer bound in rmi registry
```

To stop the server type the following command:

```
cloudscape -stop
```

You should see output similar to the following:

```
Attempting to shutdown RmiJdbc server
RmiJdbc Server RmiAddr is: //buzz/RmiJdbcServer
WARNING: Shutdown was successful!
```

Note: If you stop the server with Control-c, files will not be closed properly. When the server is started the next time, it must perform recovery by rolling back non-committed transactions and possibly applying the forward log.

Cloudscape Server Configuration

The default database used by the Cloudscape server is named CloudscapeDB. This database will reside in the `$J2EE_HOME/cloudscape` directory (where `$J2EE_HOME` is the directory where you've installed the J2EE SDK.) The CloudscapeDB database will be created automatically the first time it is accessed. The driver for the Cloudscape server is already configured in the `$J2EE_HOME/config/default.properties` file. No further changes by you are necessary.

Cloudscape ij Tool

The Cloudscape product includes an interactive SQL tool called `ij`. (This tool is not supported by Sun Microsystems, Inc.) You can run the `ij` tool by executing the `cloudIJ.sh` (UNIX) or `cloudIJ.bat` (Windows) script, which resides in the `doc/guides/ejb/examples/util` directory.

J2EE Server

To launch the J2EE server, run the `j2ee` script from the command-line prompt. The following table describes the `j2ee` script's options:

TABLE 10-2 Options of the j2ee Script

Option	Description
-verbose	Redirects all logging output to the current shell.
-version	Displays the version number.
-stop	Stops the J2EE server.
-singleVM	Runs services and deployed enterprise beans in a single process. This mode is the default. You'll probably want to use this mode when debugging your applications because debugging multiple processes can be difficult.
-multiVM	Launches an additional VM (virtual machine) for each application that you deploy. Also launches separate VMs for the EJB and HTTP services. This option may improve performance but it will increase memory usage.

To run the HTTPS service of the J2EE server, you must install a server certificate. For instructions, see the section "Setting Up a Server Certificate" on page 133.

Key Tool

The `keytool` utility creates public and private keys and generates X509 self-signed certificates. The J2EE SDK version of the `keytool` utility has the same options as the version distributed with the J2SE SDK . However, the J2EE version programatically adds a Java™ Cryptographic Extension provider that has impementations of RSA algorithms (licensed from RSA Data Security). For more information, see the section "Setting Up a Server Certificate" on page 133.

Packager

The `packager` tool is a command-line script allows you to package J2EE components. This tool is for advanced users who do not want to use the Application Deployment Tool to package J2EE components. With the `packager`, you can create the following component packages:

- EJB .jar File
- Web Component .war File
- Application Client .jar File
- J2EE Application .ear File

EJB .jar File

The syntax for packaging an EJB .jar file follows:

```
packager <root-directory> package/Class1.class:package/
Class2.class:pics/me.gif ejb-jar.xml ejb.jar
```

Example:

The following command packages the EJB classes and the ejb-jar.xml deployment descriptor into the myEjbJar.jar file:

```
packager -ejbJar /home/duke/classes/
HelloHome.classHelloEJB.class:HelloRemote.class:Util.class test/ejb-
jar.xml myEjbJar.jar
```

Web Component .war File

The syntax for packaging a web component .war file follows:

```
packager -webArchive[-classpath servletorjspbean/classes [ -
classFiles package/MyClass1.class: package/MyClass2.class ] ]
<content-root> [-contentFiles login.jsp:index.html:images/me.gif]
web.xml myWebApp.war
```

Simple Example:

The following command packages the myWebPage.xml deployment descriptor and the page in WebPageDir/Hello.html into the myWebPage.war file:

```
packager -webArchive myWebPageDir myWebPage.xml myWebPage.war
```

Specifying Individual Content Files:

Suppose that you add a Hello.jsp file to the directory myWebPageDir, don't want the Hello.html file any more, and modify your .xml file accordingly. You can individually specify the content files to add using the -contentFiles flag:

```
packager -webArchive myWebPageDir -contentFiles Hello.jsp
myWebPage.xml myWebPage.war
```

Without the -contentFiles option, the following command will produce the same .war file because it includes everything under the directory myWebPageDir:

```
packager -webArchive myWebPageDir -contentFiles Hello.jsp:Hello.html
myWebPage.xml myWebPage.war
```

Specifying Servlets and JSP Files:

Suppose that you write a servlet and compile it into your classes directory, modifying the .xml file for its deployment attributes. Its class file is classes/package/Servlet1.class. The following command includes the servlet class file because it is under the classes directory:

```
packager -webArchive -classpath classes myWebPageDir -contentFiles
Hello.jsp myWebPage.xml myWebPage.war.
```

The following command specifies that only the package/Servlet1.class and packageB/Servlet.class files are to be included:

```
packager -webArchive -classpath classes -classFiles package/
Servlet1.class:packageB/Servlet.class myWebPageDir -contentFiles
Hello.jsp myWebPage.xml myWebPage.war
```

The next command adds the Hello.html file back into the .war file:

```
packager -webArchive -classpath classes -classFiles package/
Servlet1.class:packageB/Servlet.class myWebPageDir -contentFiles
Hello.jsp:Hello.html myWebPage.xml myWebPage.war
```

Application Client .jar File

The syntax for packaging an application client .jar file follows:

```
packager -applicationClient    <root-directory> package/
Class1.class:package/Main.class:pics/me.gif package.Main client.xml
appClient.jar
```

Example:

The following command creates the appClient.jar file:

```
packager  classes hello/HelloClient.class:hello/HelloUtil.class
package.Main client.xml appClient.jar
```

J2EE Application .ear File

The syntax for packaging a J2EE application .ear file follows:

```
packager -enterpriseArchive    myWeb.war:myEJB.jar:myOtherApp.ear [-
alternativeDescriptorEntries myWeb/web.xml:myEjb/myEjb.xml: ][-
libraryJars ejblib.jar:ejblib1.jar] myAppName myApp.ear
```

Example:

In the following command, the optional -alternativeDescriptorEntries flag allows you to specify the external descriptor entry name of each component as you wish it to appear in the .ear file:

```
packager -enterpriseArchive    myWeb.war:myEJB.jar:appClient.ear -
alternativeDescriptorEntries myWeb/web.xml:myEjb/myEjb.xml:client/
client.xml myAppName myApp.ear
```

After packaging, any manipulation of the deployment information will not be written back into the component files inside the .ear file, but to the entry names in the .ear file that you specified.

Specifying the Runtime Deployment Descriptor

The preceding example specified the -enterpriseArchive flag to create a portable J2EE application .ear file. This file is portable because you can import it into any J2EE environment that conforms to the J2EE Specification. Although you can import the file into the Application Deployment Tool, you cannot deploy it on the J2EE server until it contains a runtime deployment descriptor. This deployment descriptor is an XML file that contains information such as the JNDI names of the application's enterprise beans.

In the following command, the -setRuntime flag instructs the packager to insert the runtime deployment descriptor (sun-j2ee-ri.xml) into the myApp.ear file:

```
packager -setRuntime MyApp.ear sun-j2ee-ri.xml
```

To obtain an example of the runtime deployment descriptor, extract it from a .ear file that you've already deployed:

```
jar -xvf SomeApp.ear
```

The DTD of the runtime deployment descriptor is in the `lib/dtds/sun-j2ee-ri-dtd` file of your J2EE SDK installation.

Note: The runtime deployment descriptor (`sun-j2ee-ri.xml`) is not required by the J2EE Specification. This descriptor is unique to the J2EE SDK and may change in future releases.

Realm Tool

The `realmtool` utility is a simple command-line program that allows you to add and remove J2EE users and to import certificate files. The utility has the following syntax:

```
realmtool <options>
```

The following table describes the options:

TABLE 10-3 Realm Tool Options

Option	Description
-show	Lists the realm names.
-list <realm-name>	Lists the users in the specified realm. This release has two realms: default and certificate.
-add <username password group[,group]>	Adds the specified user to the default rsealm.
-addGroup <group>	Adds a group to the default realm.

TABLE 10-3 Realm Tool Options

Option	Description
-import <certificate-file>	Adds a user to the certificate realm by importing a file containing a X509 certificate.
-remove <realm-name username>	Removes a user from the specified realm.
-removeGroup <group>	Removes a group.

Examples in the section “Managing J2EE Users and Groups” on page 125 show how to run the `realmtool` utility.

Runclient Script

To run a J2EE application client, you execute the `runclient` script. This script has the following syntax:

```
runclient -client <appjar> [-name <name>] [<app-args>]
```

The following table describes the options:

TABLE 10-4 Runclient Script Options

Option	Description
<appjar>	the J2EE application .ear file
<name>	the display name of the J2EE application client component
<app-args>	any arguments required by the J2EE application

For an example, see the section “Running the J2EE Application Client” on page 99.

Verifier

The verifier validates the following types of J2EE component files:

- J2EE application .ear
- EJB .jar
- web component .war
- application client .jar

You can run the verifier three ways:

- from within the Application Deployment Tool
- as a command-line utility
- as a stand-alone GUI utility

To run the verifier from within the Application Deployment Tool, choose Verifier from the Tools menu. The following sections explain how to run the verifier the other two ways.

Command-Line Verifier

The command-line verifier has the following syntax:

```
verifier [options] <filename>
```

The `filename` argument is the name of a J2EE component file. The following table lists the options.

TABLE 10-5 Verifier Options

Syntax	Description
-v	displays verbose version of output

TABLE 10-5 Verifier Options

Syntax	Description
-o<output-file>	writes results to <output-file>, overriding default Results.txt file
-u	runs GUI utility
-<report-level>	determines whether warnings or failures are reported, where <report-level> may be either a, w, or f: a (all results) w (warnings only) f (failures only) By default, only warnings and failures are reported.

Stand-Alone GUI Verifier

To run the stand-alone GUI verifier, follow these steps:

1. From the command-line, type:

```
verifier -u
```

2. To select a file for verification, click Add.

3. Select the radio button to indicate the report level:

- All Results
- Failures Only
- Failures and Warnings Only

4. Click OK.

5. The verifier lists the details in the lower portion of the screen.

Appendix A: Code Examples

The following table lists the code examples documented in this manual. Because the examples are meant to illustrate specific concepts, they are simple and brief. For an example of a full-scale J2EE application, see the J2EE Blueprints web site (http://java.sun.com/j2ee/blueprints/index.html).

TABLE A Documented Code Examples

Topic Demonstrated by the Example	Section Documenting the Example	Location of Source Code (subdirectory under doc/guides/ejb/examples)
simple stateless session bean	"Getting Started" on page 17	converter
stateful session bean w. exception and helper class	"A Session Bean Example" on page 29	cart
environment entry	"Accessing Environment Entries" on page 41	checker
entity bean w. bean-managed persistence	"A Bean-Managed Persistence Example" on page 44	account
entity bean w. container-managed persistence	"A Container-Managed Persistence Example" on page 58	product
entity beans w. one:one relationships	"One-to-One Relationships" on page 136	storagebin
entity beans w. one:many relationships and helper classes	"A Helper Class for the Child Table" on page 139	order

TABLE A Documented Code Examples

Topic Demonstrated by the Example	Section Documenting the Example	Location of Source Code (subdirectory under doc/guides/ejb/examples)
entity beans (2) w. one:many relationships	"An Entity Bean for the Child Table" on page 143	salesrep
entity beans w. many:many relationships	"Many-to-Many Relationships" on page 147	enroller
database connection	"How to Connect" on page 74	account
container-managed transaction	"Container-Managed Transactions" on page 80	bank
bean-managed, JDBC™ transaction	"JDBC Transactions" on page 87	warehouse
bean-managed, JTA transaction	"JTA Transactions" on page 88	teller
stand-alone Java™ application client	"Stand-Alone Java™ Applications" on page 95	converter
J2EE application client	"J2EE Application Clients" on page 96	converter
servlet client of an enterprise bean	"Servlets" on page 100	adder
JSP client w. JavaBeans™ comp.	"JavaServer Pages™ Components" on page 106	jsptobean
enterprise bean client	"Other Enterprise Beans" on page 117	shipper
JSP client w. tag extension	"Accessing Enterprise Beans Through JSP Tag Libraries" on page 155	jsptag
enterprise bean sending an email	"Sending Email from an Enterprise Bean" on page 149	confirmer
enterprise bean connecting to a URL	Section "Connecting to a URL in an Enterprise Bean" on page 9-152	urlconnect

Index

C

D

E

F

K

L

M

N

O

P

R

S

T