Beginning J2EE 1.4: From Novice to Professional

James L. Weaver, Kevin Mukhar, and Jim Crume

Apress™

Beginning J2EE 1.4: From Novice to Professional
Copyright © 2004 by James L. Weaver, Kevin Mukhar, and Jim Crume

ISBN (pbk): 1-59059-341-3

Printed and bound in the United States of America 12345678910

Additional Material: Rick Leander, Jim MacIntosh, Ron Phillips, Andrew Watt

Technical Reviewers: James L. Weaver, Kevin Mukhar, Jim Crume, Tom Marrs, Steve Anglin

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Kylie Johnston

Copy Editor: Ami Knox

Production Manager: Kari Brooks

Production Editor: Kelly Winquist

Proofreader: Katie M. Stence

Compositor: Katy Freer

Indexer: John Collin

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

8

EJB Fundamentals

So far we've discussed the user interface, business logic, and database connection aspects of developing J2EE applications. The primary mechanism discussed to this point for expressing business logic has been JavaBeans accessed from JSP and servlet code. J2EE has a powerful facility dedicated to expressing the business logic of an application, and for accessing a database using a JavaBeans-like concept. That facility is **Enterprise JavaBeans**, also know as **EJBs** for short.

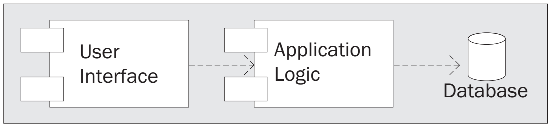
In this chapter, we'll begin exploring the world of EJBs, which is a very important capability of the J2EE platform. EJBs provide infrastructure for developing and deploying mission-critical, enterprise applications.

Throughout the chapter, you will learn:

- The benefits of using EJBs
- The three kinds of EJBs: session, entity, and message-driven beans
- What an EJB container is
- How to develop session beans
- How to use **Java Naming and Directory Interface** (**JNDI**) to locate EJBs
- Differences between stateful and stateless session beans

Understanding EJBs

Application architectures often consist of several **tiers** that each have their own responsibilities. One such architecture that consists of three tiers is illustrated in the Unified Modeling Language (UML) diagram below:



The two elements on the left-hand side of the diagram are called **components** *in the Unified Modeling Language (UML) notation. Components represent software modules. An overview of the UML is given as part of the download bundle with this book, available on the Apress web site.*

Multi-tiered, or **layered**, architectures have many advantages, not the least of which is the ability to change any one of the layers *without* affecting all of them. In the illustration above, if the *Database* layer is changed, only the *Application Logic* layer is affected. The *Application Logic* layer shields the *User Interface* layer from changes to the *Database* layer. This facilitates ongoing maintenance of the application, and increases its ability to incorporate new technologies in its layers. EJBs provide an application logic layer and a JavaBeans-like abstraction of the database layer. The application logic layer is also known as the middle tier.

JavaBeans and *Enterprise* JavaBeans are two different things, but because of their similarities (and for marketing reasons) they share a common name. JavaBeans are components built in Java that can be used on any tier in an application. They are often thought of in relationship to servlets, and as GUI components. Enterprise JavaBeans are special, server-based components used for building the business logic and data access functionality of an application.

Why Use EJBs?

Not too long ago, when system developers wanted to create an enterprise application, they would often start by "rolling their own" (or purchasing a proprietary) application server to support the functionality of the application logic layer. Some of the features of an application server include:

- ❑ **Client communication**–The client, which is often a user interface, must be able to call the methods of objects on the application server via agreed-upon protocols.
- ❑ **Session state management**–You'll recall our discussions on this topic in the context of JSP and servlet development back in Chapter 5.

- ❑ **Transaction management**–Some operations, for example when updating data, must occur as a unit of work. If one update fails, they all should fail. Recall that transactions were discussed in Chapter 7.
- ❑ **Database connection management**–An application server must connect to a database, often using **pools** of database connections for optimizing resources.
- ❑ **User authentication and role-based authorization**–Users of an application must often log in for security purposes. The functionality of an application to which a user is allowed access is often based upon the role associated with their user ID.
- ❑ **Asynchronous messaging**–Applications often need to communicate with other systems in an **asynchronous manner**, that is, without waiting on the other system to respond. This requires an underlying messaging system that provides guaranteed delivery of these asynchronous messages.
- ❑ **Application server administration**–Application servers must be **administered**. For example, they need to be monitored and tuned.

The Enterprise JavaBeans specification defines a common architecture, which has prompted several vendors to build application servers that comply with this specification. Now we can get off-the-shelf application servers that comply with a common standard, benefiting from the competition (in areas such as price, features, and performance) among those vendors. Some of the more common commercial Enterprise JavaBeans application servers are: WebLogic (BEA), Sun ONE (Sun), and WebSphere (IBM).

There are also some very good open-source entries in this market such as JBoss and JOnAS. As you know, Sun provides a freeware Reference Implementation (J2EE SDK) of the J2EE 1.4 and EJB 2.1 specifications that may be used to develop as well as to test an application for compliance with those specifications. The Reference Implementation may not, however, be used to deploy production systems.

The Sun Reference Implementation was used to develop all of the examples and exercises contained in this book.

These application servers, in conjunction with the capabilities defined in the EJB specification, support all of the features listed above and many more. Since they all support the EJB specification, we can develop full-featured enterprise applications and still avoid application server, operating system, and hardware platform vendor lock-in.

Yes, things sure have improved! We now have a standard, specifications-based way to develop and deploy enterprise-class systems. We are approaching the Java dream of developing an application that can run on any vendor platform as-is. This is in contrast to the vendor-specific way we used to develop where each server had its own way of doing things, and where the developer was locked into the chosen platform once the first line of code was written!

For more information on the EJB specification, see the http://java.sun.com/products/ejb/docs.html web site.

The Three Kinds of EJBs

As we mentioned briefly at the start of this chapter, there are actually three kinds of EJBs:

- ❑ Session beans
- ❑ Entity beans
- ❑ Message-driven beans

When referring to them in the general sense in this book, we'll use the term **EJBs**, **enterprise beans**, or simply **beans**. Here is a brief introduction to each type of bean. The balance of this chapter will then focus on **session beans**.

Session Beans

One way to think about the application logic layer (middle tier) in the example architecture described above is as a set of objects that, together, implement the business logic of an application. Session beans are the construct in EJBs designed for this purpose. In the diagram below, we see that that there may be multiple session beans in an application and each handles a subset of the application's business logic. A session bean tends to be responsible for a group of related functionality. For example, an application for an educational institution might have a session bean whose methods contain logic for handling student records. Another session bean might contain logic that maintains the lists of courses and programs available at that institution.

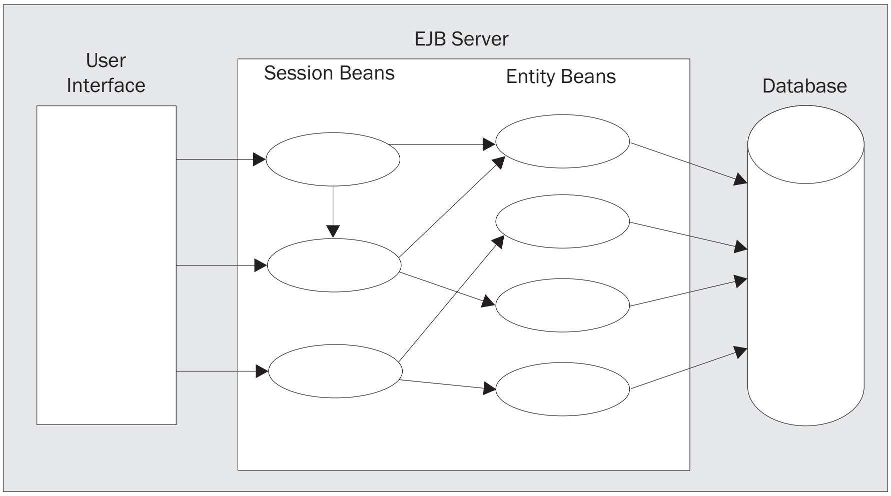
There are two types of session bean, which are defined by their use in a client interaction:

- ❑ **Stateless**–These beans do not declare any instance (class-level) variables so that the methods contained within can only act upon any local parameters. There is no way to maintain state across method calls.
- ❑ **Stateful**–These beans can hold client state across method invocations. This is possible with the use of instance variables declared in the class definition. The client will then set the values for these variables and then use these values in other method calls.

Stateless session beans provide excellent scaleability because the EJB container does not have to keep track of their state across method calls. However, storing the state of an EJB is a very resource-intensive process. There may be more work involved for the server to share stateful session beans than stateless beans. So the use of stateful beans in your application may not make it as easily scaleable as using stateless beans.

All EJBs, session beans included, operate within the context of an **EJB server**, shown in the diagram below. An **EJB server** contains constructs known as **EJB containers** that are responsible for providing an operating environment for managing and providing services to the EJBs that are running within it.

In a typical scenario, the user interface (UI) of an application calls the methods of the session beans as it requires the functionality that they provide. Session beans can call other session beans and entity beans. The diagram below illustrates typical interactions between the user interface, session beans, entity beans, and the database:



Entity Beans

Before object orientation became popular, programs were usually written in procedural languages and often employed relational databases to hold the data. Because of the strengths and maturity of relational database technology, it is now often advantageous to develop object-oriented applications that use relational databases. The problem with this approach is that there is an inherent difference between object-oriented and relational database technologies, making it less than natural for them to coexist in one application. The use of entity beans is one way to get the best of both of these worlds, because:

- Entity beans are objects, and they can be designed using object-oriented principles and utilized in applications as objects.
- The data in these entity bean objects are usually persisted in relational databases. All of the benefits of relational technologies, including maturity of products, speed, reliability, ability to recover, and ease of querying, can be leveraged.

In a typical EJB scenario, when a session bean needs to access data it calls the methods of an entity bean. Entity beans represent the persistent data in an EJB application. For example, an application for an educational institution might have an entity bean named `Student` that has one instance for every student that is enrolled in an institution. Entity beans, often “backed” by a relational database, read and write to tables in the database. Because of this, they provide an object-oriented abstraction to a relational database. Entity beans will be covered in detail in the next chapter.

Message-Driven Beans

When an EJB-based application needs to receive asynchronous messages from other systems, it can leverage the power and convenience of **message-driven beans**. Asynchronous messages between systems can be analogous to the events that are fired from a UI component to an event handler in the same JVM. One example application that could use message-driven beans is in the business to business (B2B) domain: a wholesaler could have an EJB application that uses message-driven beans to listen for purchase orders issued electronically from retailers.

Decisions, Decisions

So, how do you decide whether a given enterprise bean should be a session bean, entity bean, or a message-driven-bean? A set of rules to remember here:

- ❑ Session beans are great at implementing business logic, processes, and workflow. For example, a `StockTrader` bean with `buy()` and `sell()` methods, among others, would be a good fit for a session bean.
- ❑ Entity beans are the persistent data objects in an EJB application. In a stock trading application, a `Stock` bean with `setPrice()` and `getPrice()` methods would be an appropriate use of an entity bean. The `buy()` method of the previously mentioned `StockTrader` session bean would interact with instances of the `Stock` entity bean by calling their `getPrice()` methods for example.
- ❑ Message-driven beans are used for the special purpose of receiving asynchronous messages from other systems, like the fictitious wholesaler application mentioned above that listens for purchase orders.

By the way, as seen in the diagram above, it is a good practice to call only session beans directly from the client, and to let the session beans call the entity beans. Here are some reasons for this:

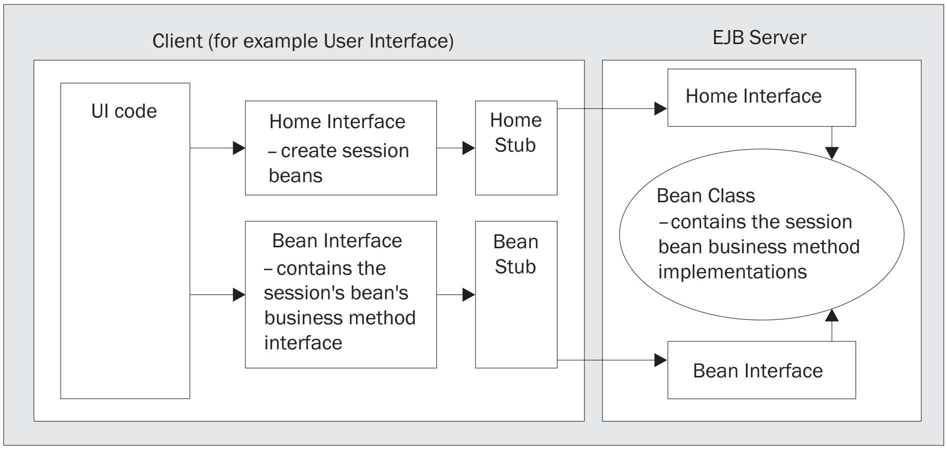
- ❑ This practice doesn't circumvent the business logic contained in the session beans. Calling entity beans directly tends to push the business logic into the UI logic, which is usually a bad thing.
- ❑ The UI doesn't have to be as dependent upon changes to the entity beans. The UI is shielded from these changes by the session beans.
- ❑ In order for a client to interact with a bean on the EJB server, there must be a remote reference to the bean, which takes resources. There tends to be far more (orders of magnitude) entity bean instances in an application than session bean instances. Restricting client access to session beans conserves server and network resources considerably.

A Closer Look at Session Beans

Now that we've covered some basics concerning the three types of EJBs, we'll use the rest of this chapter to take a closer look at the first type mentioned–session beans.

The Anatomy of a Session Bean

To develop a session bean, you actually need to create two Java interfaces and a Java class. These interfaces and the class are called the **home interface**, **bean interface**, and **bean class**, respectively. These are illustrated in the following diagram:


Client (for example User Interface)
EJB Server
UI code
Home Interface – create session beans
Home Stub
Bean Interface – contains the session's bean's business method interface
Bean Stub
Home Interface
Bean Class – contains the session bean business method implementations
Bean Interface

It is worth noting that developing entity beans also requires that you create a home interface, a bean interface, and a bean class. We'll make some entity beans in the next chapter.

The Home Interface

In order for the client of a session bean to get a reference to that bean's interface, it must use the bean's home interface. Incidentally, the home interface for an EJB extends the `EJBHome` interface of the `javax.ejb` package; the package that EJB-related classes reside in.

As a naming convention for this book, we'll append the word `Home` to the name of a bean to indicate that it is a home interface. For example, a session bean with the name StockTrader would have a home interface named `StockTraderHome`.

The Bean Interface

Session beans have an interface that exposes their business methods to clients. This bean interface extends the `EJBObject` interface of the `javax.ejb` package.

As a naming convention for this book, we'll use the name of a bean as the name of its bean interface. For example, a session bean with the name StockTrader would have a bean interface named `StockTrader`.

The Bean Class

The implementation of the business logic of a session bean is located in its bean class. The bean class of a session bean extends the `SessionBean` interface of the `javax.ejb` package.

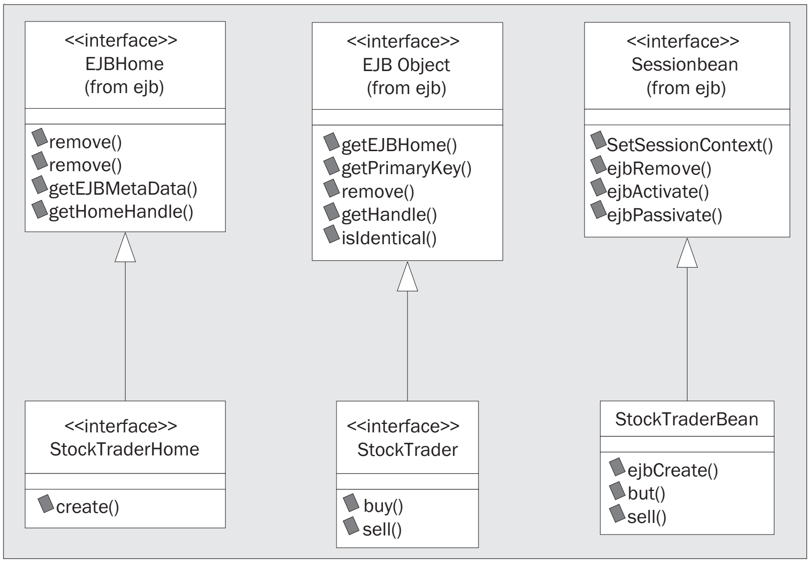
As a naming convention for this book, we'll append the word `Bean` to the name of a bean to indicate that it is a bean class. For example, a session bean with the name StockTrader would have a bean class named `StockTraderBean`.

The Home and Bean Stubs

Also in the previous diagram are the **home stub** and the **bean stub** classes. These stubs are the mechanism by which the UI code on the client can invoke methods of the EJBs that are located on the server. The stubs invoke their respective interfaces on the server side via Java Remote Method Invocation (RMI). RMI is a protocol, included in J2SE 1.4, for invoking the Java methods of a class that exists on another JVM, perhaps on a different machine.

These stubs are created for you by the Deployment Tool that has been used in this book to build and deploy JSPs and servlets. We will use it to build and deploy EJBs as well.

Here is a UML class diagram that depicts the classes, interfaces, and relationships described above:



Developing Session Beans

Well, it is now time to put all this theory into practice. In this section, we're going to develop our first session bean in an example that's on par with the traditional "Hello World!" example program.

First, we'll walk through the bean creation code in a good bit of detail, reinforcing concepts we just learned, and covering new ones. Then, we'll explain how to compile the example. For this, we'll use the Java compiler that comes with the Java 2 SDK Standard Edition 1.4 (J2SE SDK 1.4). Then, we'll show you how deploy the example. For this we'll use the Deployment Tool. Finally, we'll run the application.

Try It Out Creating a Session Bean

Since this is the first EJB example, and we haven't learned to build and deploy EJBs yet, we're going to walk through the code now and then run it later. There are four Java source files for this example:

- `SimpleSessionHome.java`
- `SimpleSession.java`
- `SimpleSessionBean.java`
- `SimpleSessionClient.java`

1. The first source file contains the code for the home interface, and should be named `SimpleSessionHome.java`. The code that this file contains should be as follows:

```
package beans;
import java.rmi.RemoteException;
import javax.ejb.EJBHome;
import javax.ejb.CreateException;

public interface SimpleSessionHome extends EJBHome {
  // The create() method for the SimpleSession bean
  public SimpleSession create()
    throws CreateException, RemoteException;
}
```

2. This is the code for the bean interface, `SimpleSession.java`:

```
package beans;

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

public interface SimpleSession extends EJBObject {
  // The public business method on the SimpleSession bean
  public String getEchoString(String clientString)
    throws RemoteException;
}
```

3. Next is the code for the bean class, `SimpleSessionBean.java`:

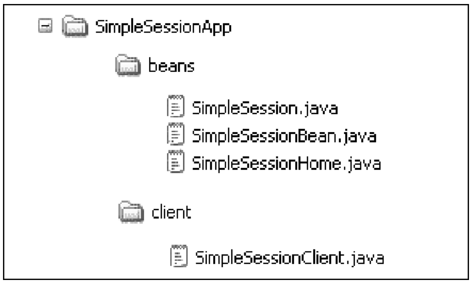
```
package beans;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
public class SimpleSessionBean implements SessionBean {
  // The public business method. This must be coded in the
  // remote interface also.
  public String getEchoString(String clientString) {
    return clientString + " - from session bean";
  }

  // Standard ejb methods
```

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

4. And this is the client code to test our session bean, `SimpleSessionClient.java`:

```
  package client;

  import beans.SimpleSession;
  import beans.SimpleSessionHome;
  import javax.naming.InitialContext;
  import javax.rmi.PortableRemoteObject;

  public class SimpleSessionClient {
    public static void main(String[] args) {
      try {
        // Get a naming context
        InitialContext jndiContext = new InitialContext();

        // Get a reference to the SimpleSession JNDI entry
        Object ref = jndiContext.lookup("ejb/beans.SimpleSession");

        // Get a reference from this to the Bean's Home interface
        SimpleSessionHome home = (SimpleSessionHome)
        PortableRemoteObject.narrow(ref, SimpleSessionHome.class);

        // Create a SimpleSession object from the Home interface
        SimpleSession simpleSession = home.create();

        // Loop through the words
        for (int i = 0; i < args.length; i++) {
          String returnedString = simpleSession.getEchoString(args[i]);
          System.out.println("sent string: " + args[i] +
                             ", received string: " + returnedString);
        }
      } catch(Exception e) {
        e.printStackTrace();
      }
    }
  }
```

These files should be organized in the following subdirectory structure:



5. Open a Command Prompt in the `SimpleSessionApp` directory.

6. Now compile the classes ensuring that the `classpath` is set to contain the `j2ee.jar` library. At the command line type:

   ```
   > set classpath=.;%J2EE_HOME%\lib\j2ee.jar
   ```

7. Within the `SimpleSessionApp` directory that the `client` and `beans` directories are located, execute the following commands from the command prompt:

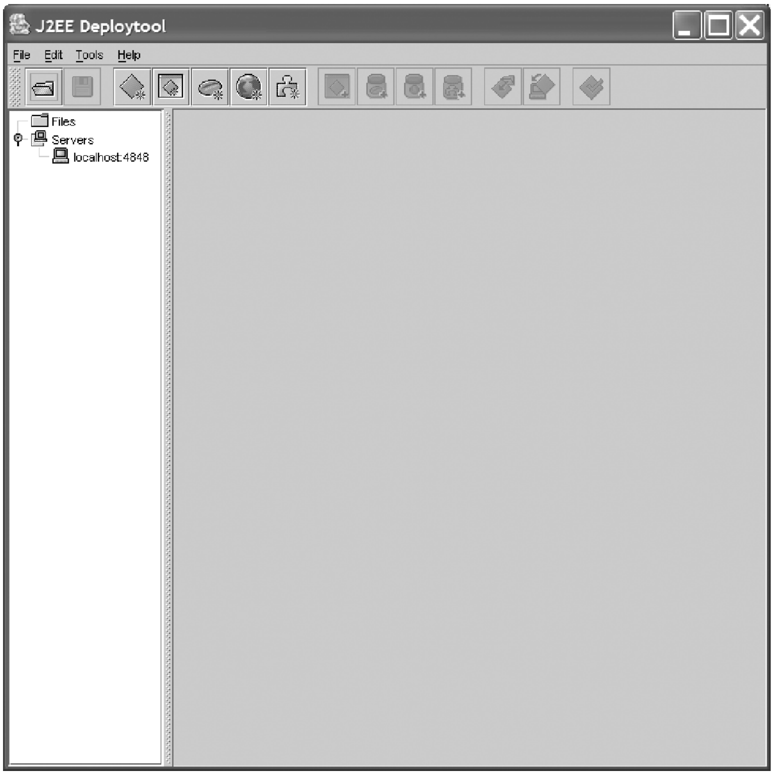
   ```
   > javac -d . client/*.java
   > javac -d . beans/*.java
   ```

   The `–d` option tells the Java compiler to place the class files in subdirectories matching their package structure, subordinate to the given directory. In this case, the given directory is the current directory, signified by the period. As a result, the Java class files should end up in the same directories as the source files.

8. Now we need to start the J2EE Server (Start Default Domain) using the instructions in Chapter 2.

9. Once the J2EE Server is up and running we need to start the Deployment Tool using the instructions in Chapter 2.

   This will display a window that looks something like this:

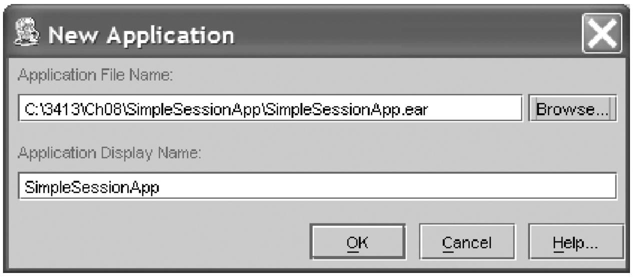


The first thing that we need the Deployment Tool to do is create the J2EE application, which will be bundled together in an **enterprise application resource** (**EAR**) file. EAR files are JAR files that contain all of the components of a J2EE application, including other JAR files and application deployment information. These other JAR files could contain:

- ❑ Enterprise beans and their deployment information
- ❑ Web application components and deployment information (recall our earlier discussion about WAR files in Chapter 7)
- ❑ Application client components and deployment information.

10. To create the application EAR file, from the File menu choose New | Application.

11. A dialog box will be displayed, prompting you to enter the name of the EAR file, and the application name that you want displayed. Let's name the EAR file SimpleSessionApp.ear, and the application display name SimpleSessionApp:



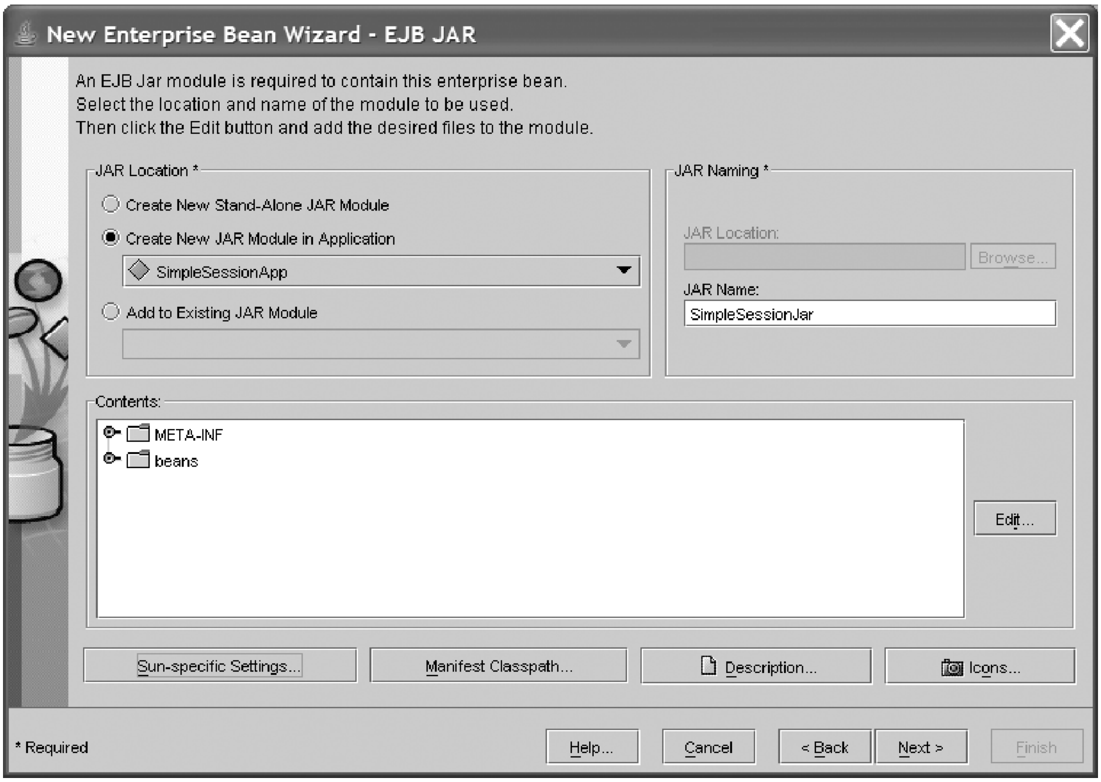
Application File Name: C:\3413\Ch08\SimpleSessionApp\SimpleSessionApp.ear

Application Display Name: SimpleSessionApp

Click OK to accept the changes to this dialog.

12. Now we'll create the JAR file in which the session bean classes and resources will be packaged. To do this, choose File | New | Enterprise Bean menu item.

13. This will start the Enterprise Bean Wizard. On the page shown below you will be asked to choose where you want the bean jar to be placed. We're going to put it in our newly created SimpleSessionApp EAR file. Also on this page is a place to enter the name for the bean jar, we will call it SimpleSessionJar. Finally, click the Edit button on the page to pick the bean class files that you want to put in the bean jar.


New Enterprise Bean Wizard - EJB JAR

An EJB Jar module is required to contain this enterprise bean.
Select the location and name of the module to be used.
Then click the Edit button and add the desired files to the module.

JAR Location *

- Create New Stand-Alone JAR Module
- [✓ Create New JAR Module in Application] SimpleSessionApp
- Add to Existing JAR Module

JAR Naming *

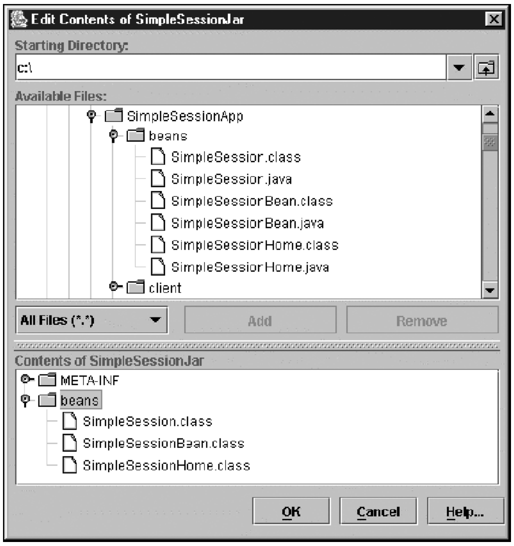
JAR Location:

JAR Name: SimpleSessionJar

Contents:

- META-INF
- beans

14. In the Available Files panel of the dialog box shown below, navigate to the beans directory of this `SimpleSessionApp` example. Choose the bean interface, the bean class, and the home interface, and click the Add button. Those bean classes will appear in the Contents of <EJB Bundle> panel as seen below:


Edit Contents of SimpleSessionJar

Starting Directory: c:\

Available Files:

```
SimpleSessionApp
  beans
    SimpleSession.class
    SimpleSession.java
    SimpleSessionBean.class
    SimpleSessionBean.java
    SimpleSessionHome.class
    SimpleSessionHome.java
  client
```

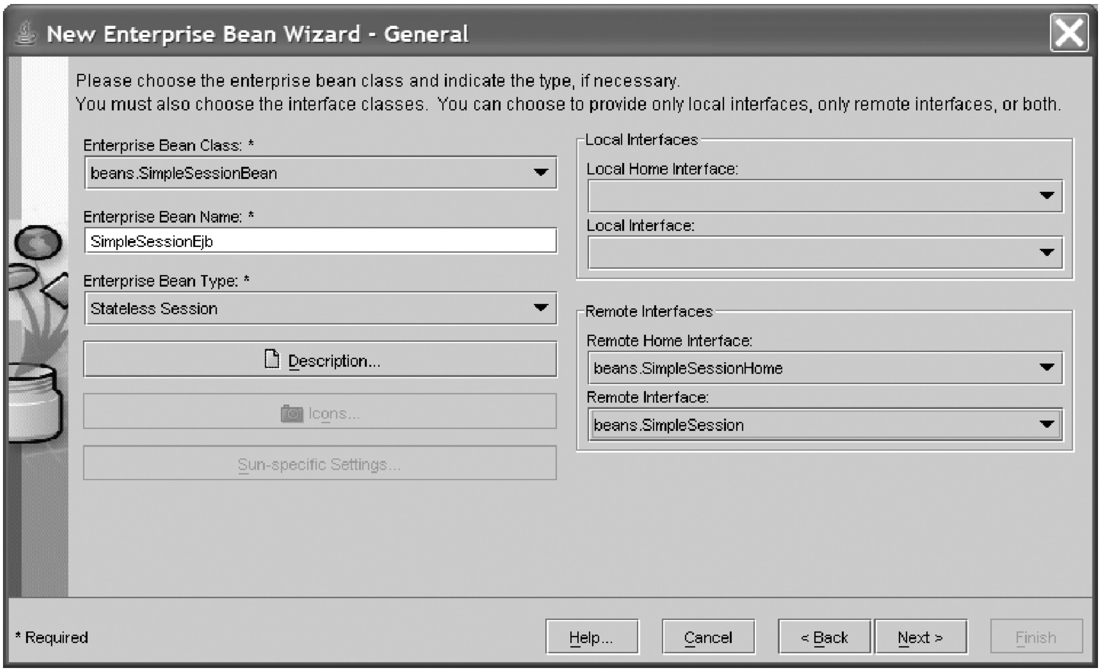
Contents of SimpleSessionJar

```
META-INF
beans
  SimpleSession.class
  SimpleSessionBean.class
  SimpleSessionHome.class
```

15. Click OK in the dialog box and then click the Next button to see the page shown below. You will then have four drop-down lists in which to make choices:

    - From the Enterprise Bean Class drop-down list, choose beans.SimpleSessionBean.
    - The bean in the `SimpleSessionApp` example is a stateless session bean. To remind you, stateless means it is not capable of storing any information within itself. We'll go into more detail about stateless session beans soon. From the Enterprise Bean Type drop-down list, go ahead and choose Stateless Session.
    - From the Remote Home Interface drop-down list, choose beans.SimpleSessionHome.
    - From the Remote Interface drop-down list, choose beans.SimpleSession.

    The Enterprise Bean Name is the name for the bean that you'd like to appear in EJB tools. The convention that we'll use is the name of the bean interface concatenated with Ejb, so enter SimpleSessionEjb into this field. When you're done with all this, your window should look like the one shown here:


New Enterprise Bean Wizard - General

Please choose the enterprise bean class and indicate the type, if necessary.
You must also choose the interface classes. You can choose to provide only local interfaces, only remote interfaces, or both.

Enterprise Bean Class: * beans.SimpleSessionBean
Enterprise Bean Name: * SimpleSessionEjb
Enterprise Bean Type: * Stateless Session

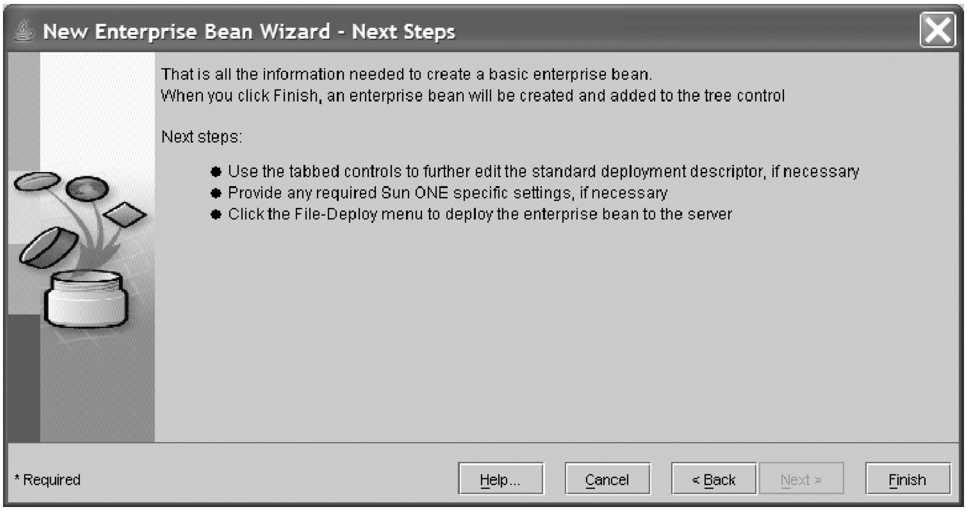
Local Interfaces
Local Home Interface:
Local Interface:

Remote Interfaces
Remote Home Interface: beans.SimpleSessionHome
Remote Interface: beans.SimpleSession

* Required

16. Click Next and the resulting dialog asks if you want the bean to be exposed as a **web service**; a concept that we'll cover in detail in Chapter 12 and Chapter 13. For this example, we don't, so just click Next.

17. The last page of the Enterprise Bean Wizard suggests some steps to do next. Click the Finish button to leave the wizard:



New Enterprise Bean Wizard - Next Steps

That is all the information needed to create a basic enterprise bean.
When you click Finish, an enterprise bean will be created and added to the tree control

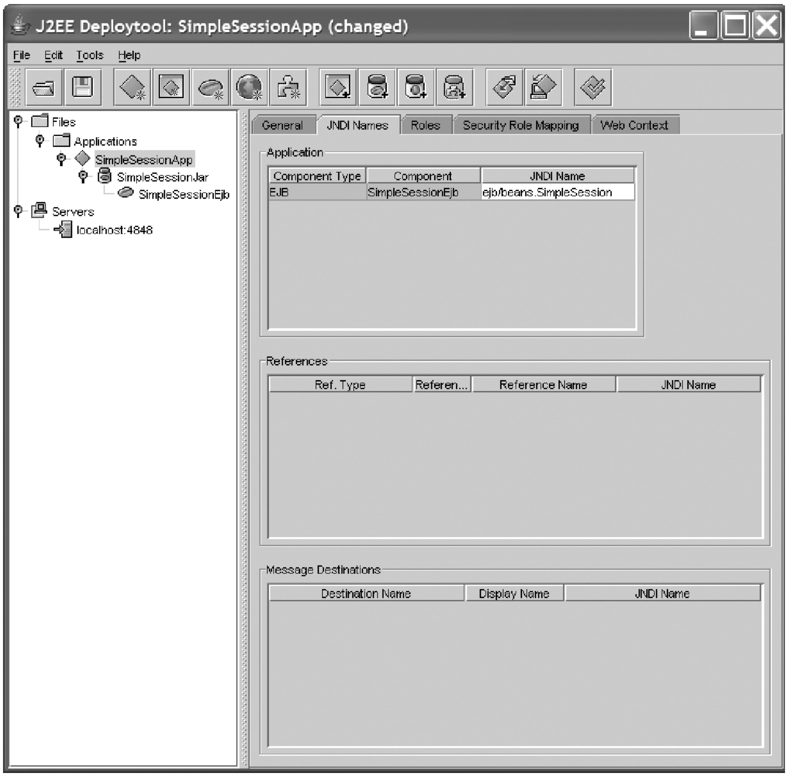
Next steps:

- Use the tabbed controls to further edit the standard deployment descriptor, if necessary
- Provide any required Sun ONE specific settings, if necessary
- Click the File-Deploy menu to deploy the enterprise bean to the server

* Required

Remember the discussion we had about JNDI and how it helped the client get a reference to the home interface of the session bean? The next step deals with this JNDI name.

18. Make sure that SimpleSessionApp is selected in the left-hand panel, as shown below. In the JNDI Name tab, type the same JNDI name that the client application uses in the `lookup()` method to obtain the bean's home reference. In this case, it is `ejb/beans.SimpleSession`.



J2EE Deploytool: SimpleSessionApp (changed)

Application

Component Type	Component	JNDI Name
EJB	SimpleSessionEjb	ejb/beans.SimpleSession

References

Ref. Type	Referen...	Reference Name	JNDI Name

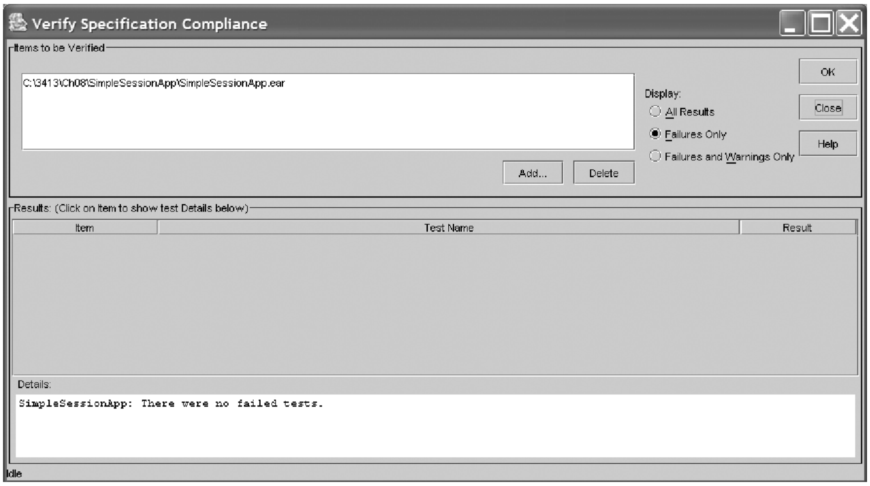
Message Destinations

Destination Name	Display Name	JNDI Name

There is one more thing to do before deploying the application: run the Verifier Tool to check whether we've configured the beans according to the EJB specifications.

19. Select the SimpleSessionApp node from the tree on the left panel and choose Verify J2EE Compliance from the Tools menu. You may be prompted to save the application.

20. To run the verification tests against the application, choose one of the Display options and click the OK button. We usually choose Failures Only option as shown below so that only the failed tests show up. The Results and the Details panels show the results of the tests and details of any problems encountered, respectively. If there are any problems encountered, then read the Details and go to the Deployment Tool page in which that detail is configured.


Verify Specification Compliance
Items to be Verified
OK
Close
Help
Add...
Delete
Results: (Click on Item to show test Details below)
Item
Test Name
Result
Idle

Items to be Verified: C:\3413\Ch08\SimpleSessionApp\SimpleSessionApp.ear

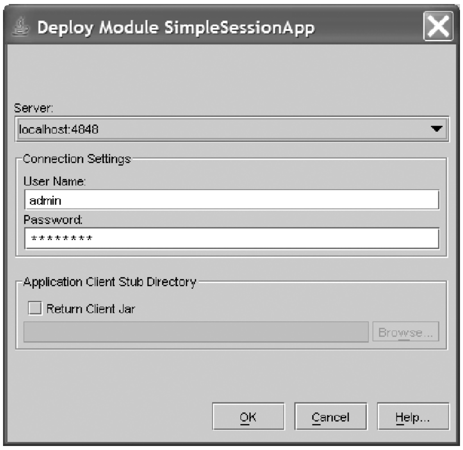
Display:
( ) All Results
(●) Failures Only
( ) Failures and Warnings Only

Details:

```
SimpleSessionApp: There were no failed tests.
```

21. If there were no failed tests, close the Verifier Tool. Then go ahead and deploy the application by selecting the SimpleSessionApp node in the tree in the left panel and selecting the Tools | Deploy menu item.

22. As a result, you should see the Deploy Module dialog, shown below. In this dialog, you are prompted for your User Name and Password:


Deploy Module SimpleSessionApp
Connection Settings
Application Client Stub Directory
Browse...
OK
Cancel
Help...

Server: localhost:4848

User Name: admin

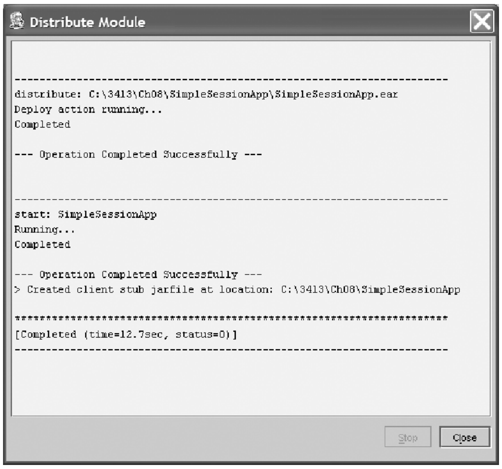
Password: ********

☐ Return Client Jar

The Deploy Module dialog also asks if you want to create a client JAR file. We need to create a client JAR file that contains the stubs that we discussed earlier. Recall that these stubs live on the client and implement the home interface and bean interface. The client can call the methods defined by those interfaces, and the stubs propagate the method invocations to the home interface and bean interface on the server.

23. To create the client JAR file, check the Return Client Jar checkbox in the dialog shown above. Enter the directory in which you want the client JAR file to be located. Choose the same directory that the client directory is rooted in. For this example, specify the name of the directory as `C:\3413\Ch08\SimpleSessionApp`. The tool will name the client JAR file `SimpleSessionAppClient.jar`, which is the name of the application's display name with "`Client.jar`" appended.

24. Click OK, and the following dialog will appear. With any luck, your bean should successfully deploy and start up, ready for clients to invoke its methods. Click the Close button when it becomes enabled:



```
---------------------------------------------------------------------
distribute: C:\3413\Ch08\SimpleSessionApp\SimpleSessionApp.ear
Deploy action running...
Completed

--- Operation Completed Successfully ---


---------------------------------------------------------------------
start: SimpleSessionApp
Running...
Completed

--- Operation Completed Successfully ---
> Created client stub jarfile at location: C:\3413\Ch08\SimpleSessionApp

*********************************************************************
[Completed (time=12.7sec, status=0)]
---------------------------------------------------------------------
```

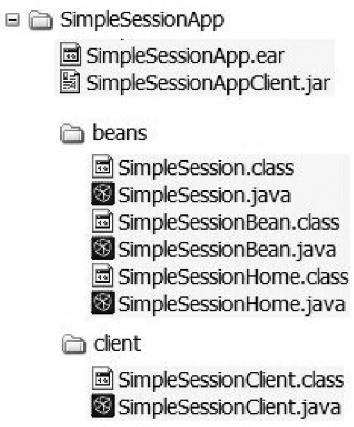
If there were any error messages when trying to deploy, please read the section called *Troubleshooting the Deploy.*

25. To see a list of the modules that are deployed in the server, select the localhost:4848 node in the Servers tree in the left panel. The right panel will display all of the deployed objects, including the SimpleSessionApp that you just deployed.

Note: As a good housekeeping measure, when you no longer need an application deployed, you should visit this page, select the deployed object, and click Undeploy.

Running the Application

The directory structure should now have the following files:



To run the example client, set the CLASSPATH to:

- The current directory (this example has used SimpleSessionApp)
- The j2ee.jar file that is in the lib directory of the Java 2 SDK, Enterprise Edition 1.4 (J2EE SDK 1.4) installation
- The appserv-rt.jar file that is in the lib directory of the Java 2 SDK, Enterprise Edition 1.4 (J2EE SDK 1.4) installation
- The SimpleSessionAppClient.jar file of the current directory

Note that it is important to use the same filename for the client JAR in the CLASSPATH as the Deployment Tool named it when creating it. In future examples, if you ever get a ClassCastException when first running the client, check to make sure that you used the same name.

1. On a default J2EE SDK 1.4 Windows installation, ensure the CLASSPATH is set correctly by using the following command:

```
> set CLASSPATH=.;%J2EE_HOME%\lib\j2ee.jar;%J2EE_HOME%\lib\appserv-
rt.jar;SimpleSessionAppClient.jar
```

2. With SimpleSessionApp as the current directory, execute the following command from the command prompt:

```
> java -Dorg.omg.CORBA.ORBInitialHost=localhost -
Dorg.omg.CORBA.ORBInitialPort=3700 client.SimpleSessionClient now is the time
```

3. When you run the `SimpleSessionClient` client program, it will produce the following output:

```
sent string: now, received string: now - from session bean
sent string: is, received string: is - from session bean
sent string: the, received string: the - from session bean
sent string: time, received string: time - from session bean
```

Not much output for all that work!

How It Works

We have four Java source files to walk through here. We'll start with the client and work our way back up to the session bean interfaces and class.

Using JNDI to Phone Home

The `main()` method of the `SimpleSessionClient` class kicks things off by using the Java Naming and Directory Interface (JNDI) to help us get a reference to the home interface of the session bean. JNDI provides a common interface to directories. The directory that we're dealing with here is internal to the EJB server and holds the reference to the home interface of our session bean. That reference is accessed using the JNDI name `ejb/beans.SimpleSession` which is the name we'll give it when we configure it using the Deployment Tool. The "`/`" and "`.`" characters are used here as separators in the JNDI name.

```
            InitialContext jndiContext = new InitialContext();

            Object ref  = jndiContext.lookup("ejb/beans.SimpleSession");
```

After we get the reference, the following statement casts the reference to type `SimpleSessionHome`:

```
            SimpleSessionHome home = (SimpleSessionHome)
            PortableRemoteObject.narrow(ref, SimpleSessionHome.class);
```

Creating and Using the Session Bean Instance

A reference to the home interface of the session bean now exists on the *client*. We use that client-held home interface to create an instance of that bean on the *server* so that its methods may be invoked. The `create()` method creates an object that implements the bean interface on the client and returns it. In this example, that reference is stored in the variable named `simpleSession`:

```
            SimpleSession simpleSession = home.create();
```

The client code for this example, which is shown below, demonstrates that we can pass an argument to a method of a session bean that exists on the server, operate on the argument in the method, and return a different value to the client. More specifically, the code loops through the arguments that were passed to the client's `main()` method via the command line, and passes them one at a time to the `getEchoString()` method of the session bean class. This is accomplished by calling the `getEchoString()` method of the bean interface that exists on the client:

```
            for (int i = 0; i < args.length; i++) {
              String returnedString = simpleSession.getEchoString(args[i]);
              System.out.println("sent string: " + args[i] +
                                 ", received string: " + returnedString);
            }
```

Note that invoking the getEchoString() method of the bean interface on the client invokes the getEchoString() method of the session bean class on the server. This is possible due to the **stub classes** described above. These stub classes are also the reason that invoking the create() method of the bean's home interface on the client was able to cause the session bean to be created on the server. When the create() method of the home interface was called using the following line of code:

```
            SimpleSession simpleSession = home.create();
```

it called the home stub class that was generated by the Deployment Tool. This home stub class implements the home interface that is defined in the SimpleSessionHome.java code listing, which we'll turn our attention to now. This interface extends the EJBHome interface located in the javax.ejb package:

```
      public interface SimpleSessionHome extends EJBHome {
```

The EJBHome interface defines three methods:

- getEJBMetaData()
- getHomeHandle()
- remove()

SimpleSessionHome defines an additional method:

```
        public SimpleSession create()
          throws CreateException, RemoteException;
```

The create() method is analogous to a constructor in a normal Java class. This particular create() method takes no arguments, but it is valid to define this method with parameters when it is desirable to pass in values at bean creation time. Like constructors, the create() method may be overloaded. When the bean is created in the EJB server, the ejbCreate() method of the bean class (SimpleSessionBean.java) will be called by the EJB container:

```
        public void ejbCreate() {}
```

In this example, the ejbCreate() method is empty, so no additional initialization will take place apart from what the EJB container will perform. Note that if we had defined a create() method with parameters in the home interface, an ejbCreate() with matching parameters would be required in the bean class.

Since SimpleSessionBean implements the SessionBean interface of the javax.ejb package, it is necessary to implement the other session bean lifecycle methods defined by that interface as well. The EJB container is responsible for calling these methods at various points in the session bean's life cycle. In this case, they are implemented with empty methods:

```
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext context) { }
```

The one and only business method in this particular session bean takes the argument passed in, appends a string of characters to it, and returns the result:

```
    public String getEchoString(String clientString) {
      return clientString + " - from session bean";
    }
```

This method is also defined in the bean interface specified in the SimpleSession.java code listing:

```
    public String getEchoString(String clientString)
      throws RemoteException;
  }
```

Note that the getEchoString() method defined in the bean interface declares that it throws a RemoteException, but the same method in the SimpleSessionBean class does need to declare that it throws that exception. This is because the business methods of the bean class are called by the EJB container, and not via RMI.

It may be helpful to refer again to the UML class diagram that depicts these classes, interfaces, and relationships. Now that we've looked at the session bean's Java source code, let's look at another source file that is necessary for session beans, the **deployment descriptor**.

About Bean Jars and Deployment Descriptors

A **bean jar** is a JAR file that is used by an EJB server, and which contains the class files for the EJBs and other resources. A **deployment descriptor** is an XML file that tells the EJB server how to deploy the beans that are in the **bean jar** file by defining their characteristics. Example characteristics include bean names, home interface names, transaction types, and bean method security access. Characteristics can be changed by editing the deployment descriptor without having to recompile the beans, which makes EJBs very flexible and maintainable. The deployment descriptor for this example was generated by the J2EE SDK 1.4 Deployment Tool, but it is possible to create and maintain it manually via a text or XML editor, and with other EJB server vendors' tools. The filename of the deployment descriptor is ejb-jar.xml, which is the EJB deployment descriptor that is portable across EJB server implementations. This XML file gets packaged and placed into the bean jar file by the Deployment Tool. The deployment descriptor for the SimpleSession example is shown below:

There are platform-specific deployment descriptors as well, which you could see by cracking open jar files that are created by various EJB tools.

```
<?xml version='1.0' encoding='UTF-8'?>
<ejb-jar
    version="2.1"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
    >
  <display-name>SimpleSessionJar</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>SimpleSessionEjb</ejb-name>
      <home>beans.SimpleSessionHome</home>
      <remote>beans.SimpleSession</remote>
      <ejb-class>beans.SimpleSessionBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
      <security-identity>
        <use-caller-identity>
        </use-caller-identity>
      </security-identity>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Since the `ejb-jar.xml` file is portable across EJB server implementations, we will examine it in conjunction with the EJB code examples to learn about configuring enterprise bean characteristics. Let's look at a few characteristics in this example.

A bean jar's `display-name` is the name that would appear in a given vendor's EJB tools, for example an EJB server administration tool:

```
  <display-name>SimpleSessionJar</display-name>
```

A bean's `ejb-name` is a unique name for that bean within the `ejb-jar.xml` file:

```
      <ejb-name>SimpleSessionEjb</ejb-name>
```

The class names of the home interface, bean interface, and bean class are specified as well:

```
      <home>beans.SimpleSessionHome</home>
      <remote>beans.SimpleSession</remote>
      <ejb-class>beans.SimpleSessionBean</ejb-class>
```

We mentioned briefly that a bean interface can be a *remote* interface or a *local* interface. This will be covered in detail in the next chapter, but for now, note that the `<remote>` element indicates that the bean interface is a remote interface. This indicates that the bean's methods can be called from outside the JVM that it resides in.

Another bean characteristic in the `ejb-jar.xml` file is the `session-type`. In this example, the `SimpleSessionBean` is `Stateless`, which means that it can't be relied upon to remember anything after each method invocation. Anyway, we'll have more to say about **stateless** (and **stateful**) session beans later in this chapter.

Troubleshooting the Deployment

We know that it's hard to believe, but occasionally your application may not deploy successfully on the first try. This is exacerbated by the fact that we're using a reference implementation deployment tool. So if you get an exception in the Distribute Module dialog after attempting deployment, there are a few things you can do to rectify the situation:

- ❑ Obviously, go back and verify that all the instructions were followed, and run the Verifier Tool.
- ❑ If you are still getting exceptions when deploying, then try the following steps until you get a successful deploy.

1. Select the applications in the tree on the left, and select Close from the File menu. Exit the Deployment tool. Stop the J2EE application (closing the J2EE window will usually accomplish this). Start the J2EE application again. Start the Deployment Tool. Open your application again by selecting the Open from the File menu and finding your application EAR file. Try to deploy again.
2. If it still doesn't deploy, then repeat the above step, rebooting your machine after stopping the J2EE application.
3. If it still doesn't deploy, then uninstall the J2EE SDK (backing up anything that you care about in the J2EE SDK directory structure) and reinstall. This may seem like a drastic measure, but we've had to take this step on rare occasions.
4. If the application still won't deploy, then it seems reasonable that a bean configuration detail has been missed or incorrectly performed.

When the bean is successfully deployed, we'll get the client ready to access it.

What Did We Learn from This?

In this `SimpleSessionApp` example, we learned how to develop a session bean, including how to deploy and start it in a J2EE application server. We also learned how to develop a client application that uses session beans. As we briefly mentioned in the deployment descriptor discussion, the `SimpleSessionBean` was deployed as a stateless session bean, which means that it can't be counted on to retain data between method invocations. The next section will introduce the idea of a **stateful** session bean, and compare these two types.

Stateful vs. Stateless Session Beans

As mentioned previously, session beans are great choice for implementing business logic, processes, and workflow. When you choose to use a session bean to implement that logic, you have yet another choice to make: whether to make that session bean **stateful** or **stateless**.

Choosing Between Stateful and Stateless Beans

Consider a fictitious stock trading application where the client uses the `buy()` and `getTotalPrice()` methods of a `StockTrader` session bean. If the user has several different stocks to buy and wants to see the running total price on the tentative purchases, then that state needs to be stored somewhere. One place to store that kind of transient information is in the instance variables of the session bean itself. This requires that the session bean be defined as stateful, as we learned previously, in the `ejb-jar.xml` (EJB deployment descriptor) file.

There are advantages for choosing that a session bean be stateful, and some for it being stateless. Some advantages of being stateful are:

- ❑ Transient information, such as that described in the stock trading scenario, can be stored easily in the instance variables of the session bean, as opposed to defining and using entity beans (or JDBC) to store it in a database.
- ❑ Since this transient information is stored in the session bean, the client doesn't have to store it and potentially waste bandwidth by sending the session bean the same information repeatedly with each call to a session bean method. This bandwidth issue is a big deal when the client is installed on a user's machine that invokes the session bean methods over a phone modem, for example. Bandwidth is also an issue when the data is very large or needs to be sent many times repeatedly.

The main disadvantage of being stateful is:

- ❑ Stateful session beans don't scale up as well on an EJB server, because they require more system resources than stateless session beans do. A couple of reasons for this are that:
  - ❑ Stateful session beans require memory to store the state.
  - ❑ Stateful session beans can be swapped in and out of memory (activated and passivated) as the EJB container deems necessary to manage server resources. The state gets stored in a more permanent form whenever a session bean is passivated, and that state is loaded back in when the bean is activated.

By the way, you may recall that the `SessionBean` interface defines several session bean lifecycle methods, including `ejbActivate()` and `ejbPassivate()`. A stateful session bean class can implement these methods to cause special processing to occur when it is activated or passivated.

Let's look at an example of using stateful session beans in the context of a device that stores state–a calculator.

Try It Out Creating a Stateful Session Bean

This example mimics some very simple operations on a calculator: adding, subtracting, and keeping a running total. Not very impressive by today's standards, but you would have paid good money for a calculator with those functions in the early 1970s! That "keeping a running total" part is what we'll be demonstrating with the help of a stateful session bean. A screenshot of the GUI client follows the instructions to build and run the example.

There are four Java source files in this example:

- ❑ `Calculator.java` (in the `beans` package)
- ❑ `CalculatorBean.java` (in the `beans` package)
- ❑ `CalculatorHome.java` (in the `beans` package)
- ❑ `CalculatorClient.java` (in the `client` package)

Listed below are the bean-related classes only. The source code for `CalculatorClient.java`, as well as the source code for all the examples in this book, may be downloaded from the Apress web site.

1. Add the following code files to a new application directory called `SimpleCalculatorApp`. Within the directory add `beans` and `client` subdirectories. Copy the code for `CalculatorClient.java` into the `client` directory.

   Here is the code for the home interface, `CalculatorHome.java`:

```
package beans;

import java.rmi.RemoteException;
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
public interface CalculatorHome extends EJBHome {
  // The create method for the Calculator bean.
  public Calculator create()
    throws CreateException, RemoteException;
}
```

   As in the previous example, we supply a no-argument `create()` method.

This is the code for the bean interface, `Calculator.java`:

```
package beans;

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

public interface Calculator extends EJBObject {
  // The public business methods on the Calculator bean
  public void clearIt() throws RemoteException;
  public void calculate(String operation, int value)
    throws RemoteException;
  public int getValue() throws RemoteException;
}
```

It defines the three business methods of the calculator.

The code for the bean class, `CalculatorBean.java`:

```
package beans;

import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class CalculatorBean implements SessionBean {
  // Holds the calculator value
  private int _value = 0;

  // The public business methods. These must be coded in the
  // remote interface also.

  // Clear the calculator
  public void clearIt() {
    _value = 0;
  }

  // Add or subtract
  public void calculate(String operation, int value)
    throws RemoteException {
    // If "+", add it
    if (operation.equals("+")) {
      _value = _value + value;
      return;
    }

    // If "-", subtract it
    if (operation.equals("-")) {
      _value = _value - value;
      return;
    }

    // If not "+" or "-", it is not a valid operation
    throw new RemoteException("Invalid Operation");
  }
```

```
    // Return the value
    public int getValue() throws RemoteException {
      return _value;
    }

    // Standard ejb methods
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void ejbCreate() {}
    public void setSessionContext(SessionContext context) { }
  }
```

2. Now compile the java files following the same instructions as in the previous example. At the command line type:

```
> set classpath=.;%J2EE_HOME%\lib\j2ee.jar
```

3. Within the `SimpleCalculatorApp` directory that the `client` and `beans` directories are located, execute the following commands from the command prompt:

```
> javac -d . client/*.java
> javac -d . beans/*.java
```

4. Now start the J2EE Server and then the Deployment Tool.

5. Create a new EAR file for the application, from the File menu choose New | Application.

6. In the New Application dialog box, name the Application File Name `SimpleCalculatorApp.ear`. As before, this file should be located in the base directory of the example. In the same New Application dialog box, make the Application Display Name SimpleCalculatorApp.

7. Now create the JAR file for the session bean classes and resources. Choose the File | New | Enterprise Bean menu item.

8. In the EJB JAR page of the Edit Enterprise Bean Wizard, make the JAR Display Name SimpleCalculatorJar. We will create the JAR in the SimpleCalculatorApp EAR file.

9. Press the Edit button in the Contents section of the page. In the Available Files dialog that appears, navigate to the beans directory and add the three Calculator bean-related classes.

10. Once these have been added, click Next.

11. You will then have four drop-down lists in which to make choices:

    - ❑ From the Enterprise Bean Class drop-down list, choose beans.CalculatorBean.
    - ❑ From the Enterprise Bean Type drop-down list, choose Stateless Session.
    - ❑ From the Remote Home Interface drop-down list, choose beans.CalculatorHome.
    - ❑ From the Remote Interface drop-down list, choose beans.Calculator.

    Enter SimpleCalculatorEjb as the Enterprise Bean Display Name, and then click Next.

12. Again leave the Configuration Options page at its default setting and click Next.
13. Click Finish on the Next Steps page.
14. Select the SimpleCalculatorApp node of the tree in the left-hand panel. Enter `ejb/beans.SimpleCalculator` in the JNDI Name column of the table that is in the JNDI Names tab.
15. Again run the Verifier Tool to check your application.
16. We can now deploy the application by selecting the SimpleCalculatorApp node in the tree in the left panel and selecting the Tools | Deploy menu item. Be sure to select the Return Client Jar checkbox and type the `C:\3413\Ch08\SimpleCalculatorApp` directory path into the text box.

After doing all of the above steps, we can now get on with running the application.

Running the Application

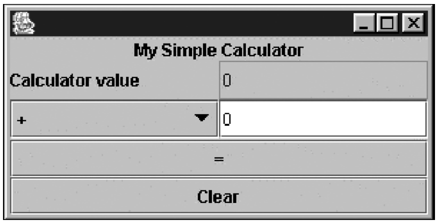
1. On a default J2EE SDK 1.4 Windows installation, the `CLASSPATH` would be set correctly by using the following command:

```
> set CLASSPATH=.;%J2EE_HOME%\lib\j2ee.jar;%J2EE_HOME%\lib\appserv-
rt.jar;SimpleCalculatorAppClient.jar
```

2. With `SimpleCalculatorApp` as the current directory, execute the following command from the operating system prompt:

```
> java -Dorg.omg.CORBA.ORBInitialHost=localhost -
Dorg.omg.CORBA.ORBInitialPort=3700 client.CalculatorClient
```

The graphical user interface (GUI) of the client should appear like this when run:



To operate the calculator, type a number into the text box, select an operation (+ or –) from the drop-down, and click the = button. The running total will be displayed beside the Calculator value label.

How It Works

To understand how this example works, we'll walk through some of the GUI client code contained in the `CalculatorClient.java` source file, and then we'll take a closer look at some of the EJB code shown above. By the way, in the code examples you'll notice that some of the `import` statements are wildcards and some explicitly name the class or interface.

For instructional purposes, we've chosen to be explicit on the imports that are relevant to J2EE, the subject of this book. We've chosen to be more frugal with lines of code by using wildcards, the more familiar ones that are relevant to J2SE.

The client is a standard Java Swing application, complete with GUI components and event handler methods. The client needs to call methods of the stateful session bean, so as in the previous example it gets a reference to the bean's home interface and creates the session bean on the server. The code that performs this is in the `getCalculator()` method of the `CalculatorClient` class, which is called from the constructor:

```
private Calculator getCalculator() {
  Calculator calculator = null;
  try {
    // Get a naming context
    InitialContext jndiContext = new InitialContext();

    // Get a reference to the Calculator JNDI entry
    Object ref  = jndiContext.lookup("ejb/beans.SimpleCalculator");

    // Get a reference from this to the Bean's Home interface
    CalculatorHome home = (CalculatorHome)
      PortableRemoteObject.narrow(ref, CalculatorHome.class);

    // Create a Calculator object from the Home interface
    calculator = home.create();
  } catch(Exception e) {
    e.printStackTrace();
  }
  return calculator;
}
```

When the = button is clicked, two things are passed to the `calculate()` method of the calculator session bean: the operator (either + or –), and the value to be added or subtracted from the running total:

```
_calculator.calculate(oper, operVal);
```

Since it is a stateful session bean, it is able to store the running total in an instance variable. The client then calls the `getValue()` method of the calculator session bean to retrieve the running total and subsequently display it:

```
_topNumber.setText("" + _calculator.getValue());
```

When the user presses the Clear button, the `clearIt()` method of the calculator session bean is called, which sets the running total to `0`.

And Now the Bean Code

The implementations of the three calculator business methods of the `CalculatorBean` class are shown below. They manipulate the instance variable named `_value`, which holds the running total between invocations of any of these calculator session bean methods.

```
      // Clear the calculator
      public void clearIt() {
        _value = 0;
      }

      // Add or subtract
      public void calculate(String operation, int value)
        throws RemoteException {
        // If "+", add it
        if (operation.equals("+")) {
          _value = _value + value;
          return;
        }

        // If "-", subtract it
        if (operation.equals("-")) {
          _value = _value - value;
          return;
        }

        // If not "+" or "-", it is not a valid operation
        throw new RemoteException("Invalid Operation");
      }

      // Return the value
      public int getValue() throws RemoteException {
        return _value;
      }
```

There are a couple of more points to take away from this example:

- ❑ There is no indication in any of the session bean code that it is stateful–that is controlled by the `ejb-jar.xml` file (deployment descriptor). An excerpt of the `ejb-jar.xml` file for the calculator stateful session bean appears below.
- ❑ A session bean that holds values in instance variables should never be configured as stateless, because the values of the instance variables are not predictable. This is because the EJB container has complete control over managing stateless (and stateful) session beans, including initializing the values of instance variables as the bean is shared among various clients. This is a common trap because sometimes the values are retained, giving a false indication that everything is OK, and then one day you can't figure out why the program isn't working correctly. From personal experience, that's a fun one to diagnose!

Indicating Stateful in the Deployment Descriptor

Here is an excerpt of the `ejb-jar.xml` file for the calculator example. Note that the `session-type` is `Stateful`:

```
       ...
  <display-name>SimpleCalculatorJar</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>SimpleCalculatorEjb</ejb-name>
      <home>beans.CalculatorHome</home>
      <remote>beans.Calculator</remote>
      <ejb-class>beans.CalculatorBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Bean</transaction-type>
      <security-identity>
        <use-caller-identity>
        </use-caller-identity>
      </security-identity>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Summary

In this chapter, we learned what Enterprise JavaBeans are, and built a case for using them. We touched on the three types of EJBs: session beans, entity beans, and message-driven beans. Then we covered when to use each type.

The balance of this chapter was then devoted to session beans, and we started that discussion by explaining that session beans are made up of three parts; the home interface, the bean interface, and the bean class. During the session bean discussions we experienced the following concepts in the context of code examples:

- Java Naming and Directory Interface (JNDI)
- Creating session beans
- Application EAR, bean jar, and client JAR files
- Deployment descriptors
- Compiling, configuring, and deploying session beans
- Using the J2EE SDK Deployment Tool to configure EJBs
- Stateless and stateful session beans

Now that we've explored session beans, in the next chapter we'll turn our attention to another type of enterprise bean–the entity bean.

Exercises

1. Write a stateless session bean that takes a word and returns it spelled backwards.
2. Write a stateful session bean that takes one word at a time and appends it to the previous words received to make a sentence. Return the entire sentence each time a word is added.
3. Modify the previous exercise, adding a stateless session bean with a method that counts the number of letters in a word. Call this method from the builder bean to count the number of letters in each word. Show this number in the returned string.