

SEPTEMBER 27, 2000

Writing Enterprise Applications with Java™ 2 SDK, Enterprise Edition

by Monica Pawlan

copyright © 1995-99 Sun Microsystems, Inc.

As used in this document, the terms “Java™ virtual machine” or “Java VM” mean a virtual machine for the Java platform.

Preface

This tutorial introduces you to the APIs, tools, and services provided in the Java™ 2 Enterprise Edition (J2EE) Software Developer Kit (SDK). You can get the free J2EE SDK (<http://java.sun.com/j2ee/download.html>) to use for demonstrations, prototyping, educational use, and verifying J2EE application portability.

To support these uses the J2EE SDK comes with J2EE development and deployment tools, a Web server, Cloudscape database, Java Software application server, Extensible Markup Language (XML) support, the J2EE APIs, and Java Plug-In. Java Plug-In lets you run Java 2 applets in browsers that support an earlier release of the Java Runtime Environment (JRE).

Note: This is a work in progress. Links to new lessons are turned on when they become available. Submit comments and suggestions to jdcee@sun.com

Contents

Preface	iii
Lesson 1	
A Simple Session Bean	1
Example Thin-Client Multitiered Application	2
J2EE Software and Setup	3
Unix:	3
Windows:	3
Path and ClassPath Settings	3
Path Settings	3
Class Path Settings	4
J2EE Application Components	4
Create the HTML Page	5
HTML Code	6
Create the Servlet	6
Import Statements	7
init Method	7
doGet Method	7
Servlet Code	9
Create the Session Bean	10
CalcHome	11
Calc	12
CalcBean	12
Compile the Session Bean and Servlet	13
Compile the Session Bean	13
Compile the Servlet	13
Start the J2EE Application Server	14
Unix:	14
Windows:	14
Start the Deploy Tool	14
Unix:	14
Windows:	14
Deploy Tool	15
Assemble the J2EE Application	16
Create J2EE Application	16
Create Session Bean	16
Create Web Component	19
Specify JNDI Name and Root Context	22

- Verify and Deploy the J2EE Application 23
- Run the J2EE Application 25
- Updating Component Code 26

Lesson 2

A Simple Entity Bean27

- Create the Entity Bean 28
 - BonusHome 28
 - Bonus 29
 - BonusBean 30
- Change the Servlet 32
- Compile 34
 - Compile the Entity Bean 34
 - Compile the Servlet 35
- Start the Platform and Tools 35
 - Unix 35
 - Windows 35
- Assemble and Deploy 35
 - Update Application File 36
 - Create Entity Bean 36
 - Verify and Deploy the J2EE Application 42
- Run the J2EE Application 43

Lesson 3

Cooperating Enterprise Beans45

- Change the Session Bean 46
 - CalcHome 46
 - Calc 47
 - CalcBean 47
- Change the Servlet 49
- Compile 50
 - Compile the Session Bean 51
 - Compile the Servlet 51
- Start the Platform and Tools 51
 - Unix 52
 - Windows 52
- Assemble the Application 52
 - Create New J2EE Application 52
 - Create New Web Component 53
 - Bundle Session and Entity Beans in one JAR File 54
- Verify and Deploy the J2EE Application 58
- Run the J2EE Application 60

Lesson 4

JavaServer Pages Technology.....61

- Create the JSP Page 62
 - Comments 64
 - Directives 64
 - Declarations 64
 - Scriptlets 65
 - Predefined Variables 65
 - Expressions 65
 - JSP-Specific Tags 66
- Change bonus.html 66
- Start the Platform and Tools 67
 - Unix 67
 - Windows 67
- Remove the WAR File 67
- Create New WAR File 67
- Verify and Deploy the J2EE Application 68
- Run the J2EE Application 70
- More Information 71

Lesson 5

Adding JavaBeans Technology to the Mix73

- About the Example 74
- Create bonus.jsp 76
 - Specify the JavaBean 78
 - Get the Data 78
 - Pass the Data to the JavaBean 78
 - Retrieve Data from the JavaBean 78
- Create the JavaBeans Class 79
 - Bean Properties 81
 - Constructor 81
 - Set Methods 81
 - Get Methods 82
- Start the Platform and Tools 84
 - Unix 84
 - Windows 84
- Remove the WAR File 85
- Create New WAR File 85
- Verify and Deploy the J2EE Application 86
- Run the J2EE Application 87
- More Information 87

Lesson 6

Extensible Markup Language (XML)89

- Marking and Handling Text 90

Change the JavaBean Class	90
XML Prolog	91
Document Root	91
Child Nodes	91
Other XML Tags	91
JavaBean Code	92
The APIs	95
SAX and DOM	95
J2EE	95
Update and Run the Application	96
More Information	96

Lesson 7

JDBC Technology and Bean-Managed Persistence 97

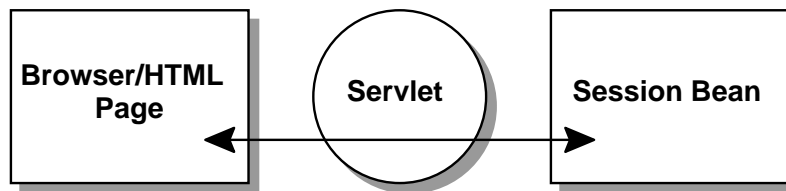
Bean Lifecycle	98
Change the BonusBean Code	99
Import Statements	99
Instance Variables	100
Business Methods	100
LifeCycle Methods	100
Change the CalcBean and JBonusBean Code	106
Create the Database Table	107
createTable.sql	107
cloudTable.bat	108
cloudTable.sh	108
Remove the JAR File	109
Verify and Deploy the Application	111
Run the Application	112
More Information	113

Index	115
--------------------	------------

Lesson 1

A Simple Session Bean

This lesson introduces you to J2EE applications programming, and the J2EE SDK by showing you how to write a simple thin-client multitiered enterprise application that consists of an HTML page, servlet, and session bean.



The J2EE SDK is a non-commercial operational definition of the J2EE platform and specification made freely available by Sun Microsystems for demonstrations, prototyping, and educational uses. It comes with the J2EE application server, Web server, database, J2EE APIs, and a full-range of development and deployment tools. You will become acquainted with many of these features and tools as you work through the lessons in this tutorial.

- Example Thin-Client Multitiered Application (page 2)
- J2EE Software and Setup (page 3)
- Path and ClassPath Settings (page 3)
- J2EE Application Components (page 4)
- Create the HTML Page (page 5)
- Create the Servlet (page 6)
- Create the Session Bean (page 10)
- Compile the Session Bean and Servlet (page 13)
- Start the J2EE Application Server (page 14)
- Start the Deploy Tool (page 14)
- Deploy Tool (page 15)
- Assemble the J2EE Application (page 16)
- Verify and Deploy the J2EE Application (page 23)
- Run the J2EE Application (page 25)
- Updating Component Code (page 26)

Example Thin-Client Multitiered Application

The example thin-client multitiered application for this lesson accepts user input through an HTML form that invokes a servlet. The servlet uses Java Naming and Directory Interface™ (JNDI) APIs to look up a session bean to perform a calculation on its behalf. Upon receiving the results of the calculation, the servlet returns the calculated value to the end user in an HTML page.

This example is a thin-client application because the servlet does not execute any business logic. The simple calculation is performed by a session bean executing on the J2EE application server. So, the client is thin because it does not handle the processing; the session bean does.

Multitiered applications can consist of 3 or 4 tiers. As shown in Figure 1, the multitiered example for this tutorial has four tiers. Three-tiered architecture extends the standard two-tier client and server model by placing a multithreaded application server between the non-web-based client application and a backend database. Four-tiered architecture extends the three-tier model by replacing the client application with a Web browser and HTML pages powered by servlet/JavaServer Pages™ technology.

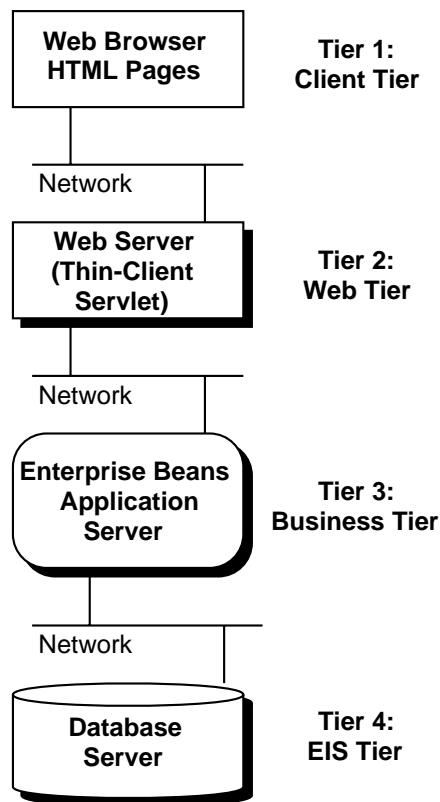


Figure 1 Multitiered Architecture

While this lesson uses only three of the four tiers, Lesson 2 expands this same example to access the database server in the fourth tier. Later lessons adapt the example to use JavaServer™ Pages and Extensible Markup Language (XML) technologies.

J2EE Software and Setup

To run the tutorial examples, you need to download and install the Java 2 SDK Enterprise Edition (J2EE), Version 1.2.1 Release (<http://java.sun.com/j2ee/download.html>), and Java 2 SDK, Standard Edition (J2SE), Version 1.2 or later (<http://java.sun.com/jdk/index.html>).

The instructions in this tutorial assume J2EE and J2SE are both installed in a J2EE directory under monicap's home directory.

Note: Everywhere `monicap` is used in a path name, please change it to your own user name.

Unix:

```
/home/monicap/J2EE/j2sdkee1.2.1  
/home/monicap/J2EE/jdk1.2.2
```

Windows:

```
\home\monicap\J2EE\j2sdkee1.2.1  
\home\monicap\J2EE\jdk1.2.2
```

Path and ClassPath Settings

The download has the J2EE application server, Cloudscape database, a Web server using secure socket layer (SSL) also known as HTTP over HTTPS, development and deployment tools, and the Java APIs for the Enterprise. To use these features, set your path and class path environment variables as described here.

Path Settings

Path settings make the development and deployment tools accessible from anywhere on your system. Make sure you place these path settings before any other paths you might have for other older JDK installations.

Unix:

```
/home/monicap/J2EE/jdk1.2.2/bin  
/home/monicap/J2EE/j2sdkee1.2.1/bin
```

Windows:

```
\home\monicap\J2EE\jdk1.2.2\bin  
\home\monicap\J2EE\j2sdkee1.2.1\bin
```

Class Path Settings

Class path settings tell the Java 2 development and deployment tools where to find the various class libraries they use.

Unix:

```
/home/monicap/J2EE/j2sdkee1.2.1/lib/j2ee.jar
```

Windows:

```
\home\monicap\J2EE\j2sdkee1.2.1\lib\j2ee.jar
```

J2EE Application Components

J2EE applications programmers write J2EE application components. A J2EE component is a self-contained functional software unit that is assembled into a J2EE application and interfaces with other application components. The J2EE specification defines the following application components:

- Application client components
- Enterprise JavaBeans components
- Servlets and JavaServer Pages components (also called Web components)
- Applets

In this lesson, you create a J2EE application and two J2EE components: a servlet and session bean. The servlet is bundled with its HTML file into a Web Archive (WAR) file, and the session bean interfaces and classes are bundled into a JAR file. The WAR and JAR files are added to the J2EE application and bundled into an Enterprise Archive (EAR) file for verification testing and deployment to the production environment.

While you do all of these steps for this lesson, you are actually performing several different functions. Writing the servlet and session bean code is a developer function, while creating a J2EE application and adding J2EE components to an application assembly function. In reality, these functions would be performed by different people in different companies.

Create the HTML Page

The HTML page for this lesson is called `bonus.html`. It's HTML code is after Figure 2, which shows how the HTML page looks when displayed to the user. The `bonus.html` file has two data fields so the user can enter a social security number and a multiplier. When the user clicks the `Submit` button, `BonusServlet` retrieves the end user data, looks up the session bean, and passes the user data to the session bean. The session bean calculates a bonus and returns the bonus value to the servlet. The servlet then returns another HTML page with the bonus value for the end user to view.

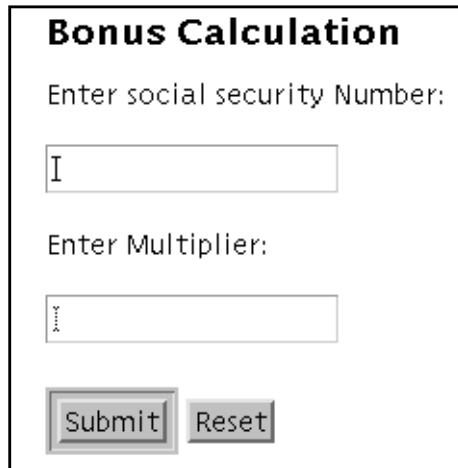


Figure 2 HTML Page

Figure 3 shows how data flows between the browser and the session bean. The session bean executes in the J2EE application server.

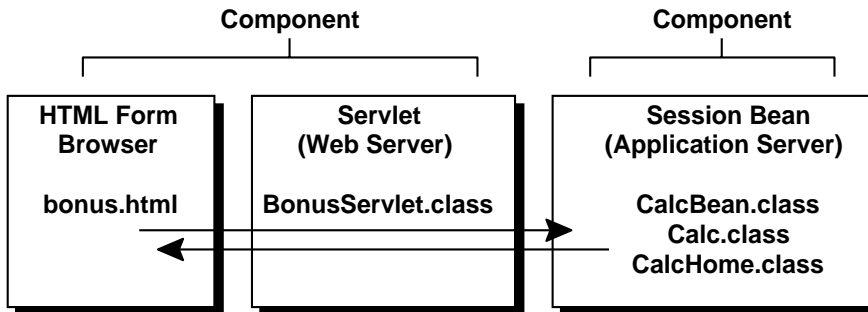


Figure 3 Data Flow

HTML Code

The interesting thing about the HTML form code is the alias used to invoke `BonusServlet`. When the user clicks the Submit button on the HTML form, `BonusServlet` is invoked because it is mapped to the `BonusAlias` during application assembly described in Assemble the J2EE Application (page 16).

The example assumes `bonus.html` is in the `/home/monicap/J2EE/ClientCode` directory on Unix. Here and hereafter, Windows users can reverse the slashes to get the correct directory pathname for their platform.

```
<HTML>
<BODY BGCOLOR = "WHITE">
<BLOCKQUOTE>
<H3>Bonus Calculation</H3>
<FORM METHOD="GET"
      ACTION="BonusAlias">
<P>
Enter social security Number:
<P>
<INPUT TYPE="TEXT" NAME="SOCSEC"></INPUT>
<P>
Enter Multiplier:
<P>
<INPUT TYPE="TEXT" NAME="MULTIPLIER"></INPUT>
<P>
<INPUT TYPE="SUBMIT" VALUE="Submit">
<INPUT TYPE="RESET">
</FORM>
</BLOCKQUOTE>
</BODY>
</HTML>
```

Create the Servlet

The example assumes the `BonusServlet.java` file is in the `/home/monicap/J2EE/ClientCode` directory on Unix. At run time, the servlet code does the following:

- Retrieves the user data
- Looks up the session bean
- Passes the data to the session bean
- Upon receiving a value back from the session bean, creates an HTML page to display the returned value to the user.

The next sections describe the different parts of the servlet code. The servlet code is shown in its entirety in Servlet Code (page 9).

Import Statements

The servlet code begins with import statements for the following packages:

- `javax.servlet`, which contains generic (protocol-independent) servlet classes. The `HttpServlet` class uses the `ServletException` class in this package to indicate a servlet problem.
- `javax.servlet.http`, which contains HTTP servlet classes. The `HttpServlet` class is in this package.
- `java.io` for system input and output. The `HttpServlet` class uses the `IOException` class in this package to signal that an input or output exception of some kind has occurred.
- `javax.naming` for using the Java Naming and Directory Interface (JNDI™) APIs to look up the session bean home interface.
- `javax.rmi` for looking up the session bean home interface and making its remote server object ready for communications.

init Method

The `BonusServlet.init` method looks up the session bean home interface and creates its instance. The method uses the JNDI name specified during component assembly (`calcs`) to get a reference to the home interface by its name. The next line passes the reference and the home interface class to the `PortableRemoteObject.narrow` method to be sure the reference can be cast to type `CalcHome`.

```
InitialContext ctx = new InitialContext();
Object objref = ctx.lookup("calcs");
homecalc = (CalcHome)PortableRemoteObject.narrow(obj
    ref, CalcHome.class);
```

doGet Method

The parameter list for the `doGet` method takes a `request` and `response` object. The browser sends a request to the servlet and the servlet sends a response back to the browser. The method implementation accesses information in the `request` object to find out who made the request, what form the request data is in, and which HTTP headers were sent, and uses the `response` object to create an HTML page in response to the browser's request.

The `doGet` method throws an `IOException` if there is an input or output problem when it handles the request, and a `ServletException` if the request could not be handled. To calculate the bonus value, the `doGet` method creates the home interface and calls its `calcBonus` method.

```
public void doGet (HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    String socsec = null;
    int multiplier = 0;
    double calc = 0.0;
    PrintWriter out;
    response.setContentType("text/html");
    String title = "EJB Example";
    out = response.getWriter();
    out.println("<HTML><HEAD><TITLE>");
    out.println(title);
    out.println("</TITLE></HEAD><BODY>");

    try{
//Retrieve Bonus and Social Security Information
        String strMult = request.getParameter(
            "MULTIPLIER");
        Integer integerMult = new Integer(strMult);
        multiplier = integerMult.intValue();
        socsec = request.getParameter("SOCSEC");

//Calculate bonus
        double bonus = 100.00;
        theCalculation = homecalc.create();
        calc = theCalculation.calcBonus(
            multiplier, bonus);
    }catch(Exception CreateException){
        CreateException.printStackTrace();
    }

//Display Data
    out.println("<H1>Bonus Calculation</H1>");
    out.println("<P>Soc Sec: " + socsec + "<P>");
    out.println("<P>Multiplier: " +
        multiplier + "<P>");
    out.println("<P>Bonus Amount: " + calc + "<P>");
    out.println("</BODY></HTML>");
    out.close();
}
```


Servlet Code

Here is the full code.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import Beans.*;

public class BonusServlet extends HttpServlet {
    CalcHome homecalc;

    public void init(ServletConfig config)
        throws ServletException{

//Look up home interface
    try{
        InitialContext ctx = new InitialContext();
        Object objref = ctx.lookup("calcs");
        homecalc =
            (CalcHome)PortableRemoteObject.narrow(
                objref,
                CalcHome.class);
    } catch (Exception NamingException) {
        NamingException.printStackTrace();
    }
}

    public void doGet (HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        String socsec = null;
        int multiplier = 0;
        double calc = 0.0;
        PrintWriter out;
        response.setContentType("text/html");
        String title = "EJB Example";
        out = response.getWriter();
        out.println("<HTML><HEAD><TITLE>");
        out.println(title);
        out.println("</TITLE></HEAD><BODY>");
        try{
            Calc theCalculation;
//Get Multiplier and Social Security Information
            String strMult =
                request.getParameter("MULTIPLIER");
            Integer integerMult = new Integer(strMult);
            multiplier = integerMult.intValue();
            socsec = request.getParameter("SOCSEC");
//Calculate bonus
```

```

    double bonus = 100.00;
    theCalculation = homecalc.create();
    calc =
        theCalculation.calcBonus(multiplier, bonus);
} catch(Exception CreateException){
    CreateException.printStackTrace();
}
//Display Data
out.println("<H1>Bonus Calculation</H1>");
out.println("<P>Soc Sec: " + socsec + "<P>");
out.println("<P>Multiplier: " +
    multiplier + "<P>");
out.println("<P>Bonus Amount: " + calc + "<P>");
out.println("</BODY></HTML>");
out.close();
}
public void destroy() {
    System.out.println("Destroy");
}
}

```

Create the Session Bean

A session bean represents a transient conversation with a client. If the server or client crashes, the session bean and its data are gone. In contrast, entity beans are persistent and represent data in a database. If the server or client crashes, the underlying services ensure the entity bean data is saved.

Because the enterprise bean performs a simple calculation at the request of `BonusServlet`, and the calculation can be reinitiated in the event of a crash, it makes sense to use a session bean in this example.

Figure 4 shows how the servlet and session bean application components work as a complete J2EE application once they are assembled and deployed. The container, shown in the shaded box, is the interface between the session bean and the low-level platform-specific functionality that supports the session bean. The container is created during deployment.

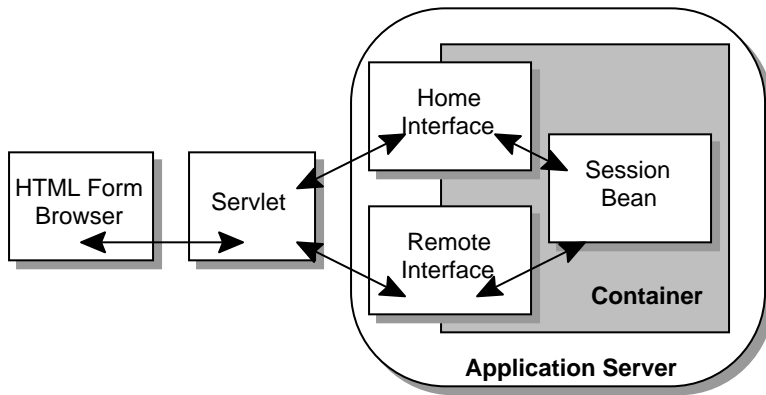


Figure 4 Application Components

The next sections show the session bean code. The example assumes the `CalcBean.java`, `Calc.java`, and `CalcHome.java` files are placed in the `/home/monicap/J2EE/Beans` directory on Unix. The `package Beans` statement at the top of the `CalcBean` interface and class files is the same name as the name of this directory. When these files are compiled, they are compiled from the directory above `Beans` and the `Beans` package (or directory) name is prepended with a slash to the interface and class files being compiled. See *Compile the Session Bean* (page 13).

Note: While this example shows how to write the example session bean, it is also possible to purchase enterprise beans from a provider and assemble them into a J2EE application.

CalcHome

`BonusServlet` does not work directly with the session bean, but creates an instance of its home interface. The home interface extends `EJBHome` and has a `create` method for creating the session bean in its container. `CreateException` is thrown if the session bean cannot be created, and `RemoteException` is thrown if a communications-related exception occurs during the execution of a remote method.

```
package Beans;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface CalcHome extends EJBHome {
    Calc create() throws CreateException,
                RemoteException;
}
```

Calc

When the home interface is created, the J2EE application server creates the remote interface and session bean. The remote interface extends `EJBObject` and declares the `calcBonus` method for calculating the bonus value. This method is required to throw `javax.rmi.RemoteException`, and is implemented by the `CalcBean` class.

```
package Beans;

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

public interface Calc extends EJBObject {
    public double calcBonus(int multiplier,
                           double bonus)
                           throws RemoteException;
}
```

CalcBean

The session bean class implements the `SessionBean` interface and provides behavior for the `calcBonus` method. The `setSessionContext` and `ejbCreate` methods are called in that order by the container after `BonusServlet` calls the `create` method in `CalcHome`.

The empty methods are from the `SessionBean` interface. These methods are called by the bean's container. You do not have to provide behavior for these methods unless you need additional functionality when the bean is, for example, created or removed from its container.

```
package Beans;

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

public class CalcBean implements SessionBean {
    public double calcBonus(int multiplier,
                           double bonus) {
        double calc = (multiplier*bonus);
        return calc;
    }
    //These methods are described in more
    //detail in Lesson 2
    public void ejbCreate() { }
    public void setSessionContext(
        SessionContext ctx) { }
    public void ejbRemove() { }
    public void ejbActivate() { }
    public void ejbPassivate() { }
    public void ejbLoad() { }
    public void ejbStore() { }
}
```

Compile the Session Bean and Servlet

To save on typing, the easiest way to compile the session bean and servlet code is with a script (on Unix) or a batch file (on Windows).

Compile the Session Bean

Unix

```
#!/bin/sh
cd /home/monicap/J2EE
J2EE_HOME=/home/monicap/J2EE/j2sdkee1.2.1
CPATH=.:$J2EE_HOME/lib/j2ee.jar
javac -d . -classpath "$CPATH" Beans/CalcBean.java
        Beans/CalcHome.java Beans/Calc.java
```

Windows

```
cd \home\monicap\J2EE
set J2EE_HOME=\home\monicap\J2EE\j2sdkee1.2.1
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar
javac -d . -classpath %CPATH% Beans/CalcBean.java
        Beans/CalcHome.java Beans/Calc.java
```

Compile the Servlet

Unix

```
#!/bin/sh
cd /home/monicap/J2EE/ClientCode
J2EE_HOME=/home/monicap/J2EE/j2sdkee1.2.1
CPATH=.:$J2EE_HOME/lib/j2ee.jar:
        /home/monicap/J2EE
javac -d . -classpath "$CPATH" BonusServlet.java
```

Windows

```
cd \home\monicap\J2EE\ClientCode
set J2EE_HOME=\home\monicap\J2EE\j2sdkee1.2
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar;
        \home\monicap\J2EE
javac -d . -classpath %CPATH% BonusServlet.java
```

Start the J2EE Application Server

You need to start the J2EE application server to deploy and run the example. The command to start the server is in the `bin` directory under your J2EE installation. If you have your path set to read the `bin` directory, go to the `J2EE` directory (so your live version matches what you see in this text) and type:

```
j2ee -verbose
```

Note: Sometimes the J2EE server will not start if Outlook is running.

If that does not work, type the following from the `J2EE` directory:

Unix:

```
j2sdkee1.2.1/bin/j2ee -verbose
```

Windows:

```
j2sdkee1.2.1\bin\j2ee -verbose
```

The `verbose` option prints informational messages to the command line as the server starts up. When you see `J2EE server startup complete`, you can start the depoloyer tool. For now, you can ignore the other messages that scrolled by.

Start the Deploy Tool

To assemble and deploy the J2EE application, you have to start the deploy tool. If you have your path set to read the `bin` directory, go to the `J2EE` directory (so your live version matches what you see in this text) and type:

```
deploytool
```

If that does not work, do the following from the `J2EE` directory:

Unix:

```
j2sdkee1.2.1/bin/deploytool
```

Windows:

```
j2sdkee1.2.1\bin\deploytool
```

Notes: If a memory access error is encountered when starting `deploytool`, add an environment variable called `JAVA_FONTS` and set the path to `c:\`. For example `c:\winnt\fonts`. Also, If a `NullPointerException` for `BasicFi-`

leChooserUI is encountered when starting `deploytool`, be sure you are not starting the tool from the root directory (i.e. `c:\`). If you run it somewhere else, such as the `bin` directory for your `j2sdkee1.2` installation, you will not encounter the problem.

Deploy Tool

The Deploy tool shown in Figure 5 has four main windows. The Local Applications window displays J2EE applications and their components. The Inspecting window displays information on the selected application or components. The Servers window tells you the application server is running on the local host. And the Server Applications window tells you which applications have been installed. As you go through the steps to assemble the example J2EE application, you will see the Local Applications, Inspecting, and Server Applications windows display information.

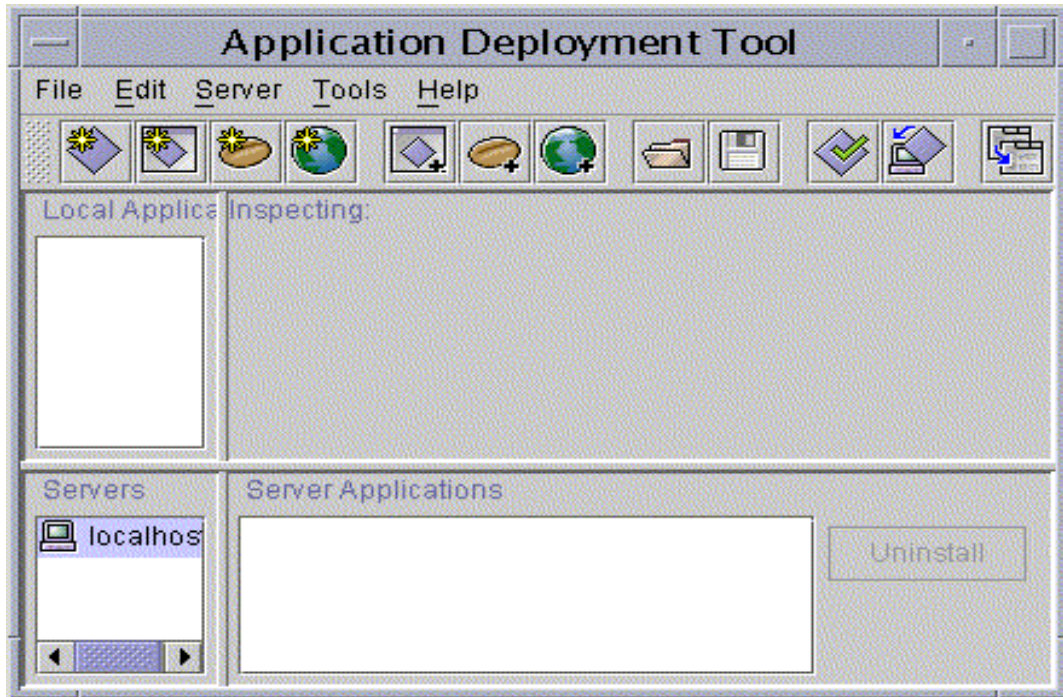


Figure 5 Deploy Tool

Note: To the right of the Server Applications window is a grayed `Uninstall` button. After you deploy the application, you will see the application listed in the Server Applications window. You can click `Uninstall` to uninstall the application, make changes, and redeploy it without having to stop and restart the application server.

Assemble the J2EE Application

Assembling a J2EE application involves creating a new application, and adding the application components to it. Here is a summary of the assembly steps, which are discussed in more detail below.

1. Create a new J2EE application (`BonusApp.ear`).
2. Create a new enterprise bean (`CalcBean.jar`).
3. Create a new web component (`Bonus.war`).
4. Specify JNDI name for the enterprise bean (`calcs`).
5. Specify the Root Context for the J2EE application (`BonusRoot`).

Create J2EE Application

J2EE components are assembled into J2EE application Enterprise Archive (EAR) files.

File menu: Select **New Application**.

New Application dialog box,:

- Type `BonusApp.ear` for the **Application File Name**.
- Click the right mouse button in the **Application Display Name** field. `BonusApp` appears as the display name.
- Click the **Browse** button to open the file chooser to select the location where you want the application EAR file to be saved.

New Application file chooser:

- Locate the directory where you want to place the application EAR file
- In this example, that directory is `/home/monicap/J2EE`.
- In the **File name** field, type `BonusApp.ear`.
- Click **New Application**.
- Click **OK**.

The `BonusApp` display name is now listed in the Local Applications window, and the Inspector window to the right shows the display name, location, and contents information for `BonusApp`. The meta information shown in the contents window describes the JAR file and J2EE application, and provides runtime information about the application.

Create Session Bean

Enterprise beans (entity and session beans) are bundled into a Java Archive (JAR) file.

File menu: Select **New Enterprise Bean**. The **New Enterprise Bean Wizard** starts and displays an **Introduction** dialog box that summarizes the steps you are about to take. After reading it over, click `Next`.

EJB JAR dialog box: Specify the following information:

- **Enterprise Bean will go in:** BonusApp
Display name: CalcJar
Description: A simple session bean that calculates a bonus. It has one method
- Click `Add`. There are two `Add` buttons on this screen. Make sure you click the second one down that is next to the **Contents** window.

Add Files to .JAR dialog box: go to the `J2EE` directory. You can either type the path name or use the browser to get there. Once at the `J2EE` directory, double click on `beans` to display the contents of the `beans` directory.

- Select `Calc.class`.
- Click `Add`.
- Select `CalcHome.class`.
- Click `Add`.
- Select `CalcBean.class`.
- Click `Add`.

Important Note: The **Add Contents to .JAR** dialog box should look like the one in Figure 6. The **Enterprise Bean JAR** classes must show the `Beans` directory prefixed to the class names.

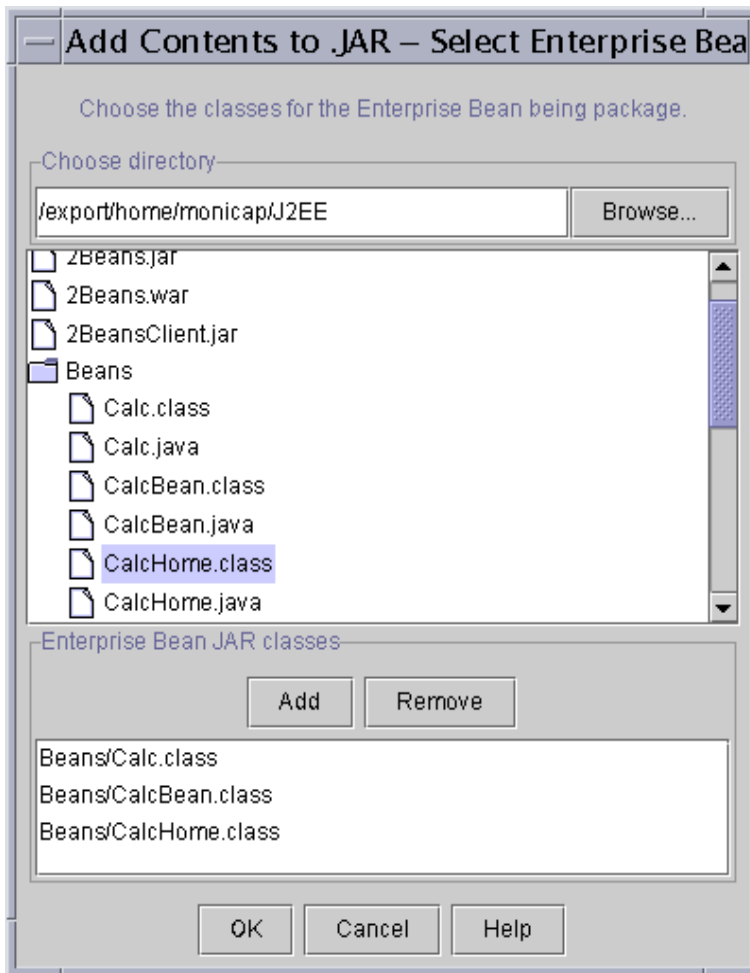


Figure 6 Select Session Bean Class Files

- Click OK. You should now be back at the **EJB JAR** dialog box. Beans/Calc.class, Beans/CalcHome.class, and Beans/CalcBean.class should appear in the **Contents** window.
- Click Next.

General dialog box: Make sure the following information is selected:

- **classname:** Beans.CalcBean
Home interface: Beans.CalcHome
Remote interface: Beans.Calc
Bean type: Session and Stateless
- Specify the display name (the name that appears when the JAR file is added to BonusApp in the Local Applications window), and provide a description of the JAR file contents.

- **Display Name:** CalcBean

- **Description:** This JAR file contains the CalcBean session bean.

- Click Next.

Environment Entries dialog box: This example does not use properties (environment entries) so you can:

- Click Finish.

Verify the JAR file was indeed added to the J2EE application:

- Go to the Local Applications window
- Click the key graphic in front of the BonusApp. You will see the CalcJar JAR file.
- Click the key graphic in front of the CalcJar to see the CalcBean session bean.

Create Web Component

Web components (servlets, or JavaServer Pages™ technology) are bundled into a Web Archive (WAR) file.

File menu: Select **New Web Component**. The **New Web Component Wizard** starts and displays a window that summarizes the steps you are about to take. After reading it over, click Next.

WAR File General Properties dialog box: Provide the following information:

- **WAR file:** BonusApp
Display name: BonusWar
Description: This war file contains a servlet and an html page.
- Click Add.

Add Contents to WAR dialog box:

- Go to the ClientCode directory by typing ClientCode after J2EE in the **Root Directory** field.
- Select bonus.html. Make sure the **WAR contents** shows the listing as bonus.html without the ClientCode directory prefixed to the name.
- Click Add.

Note: Make sure you add bonus.html before you add BonusServlet.class

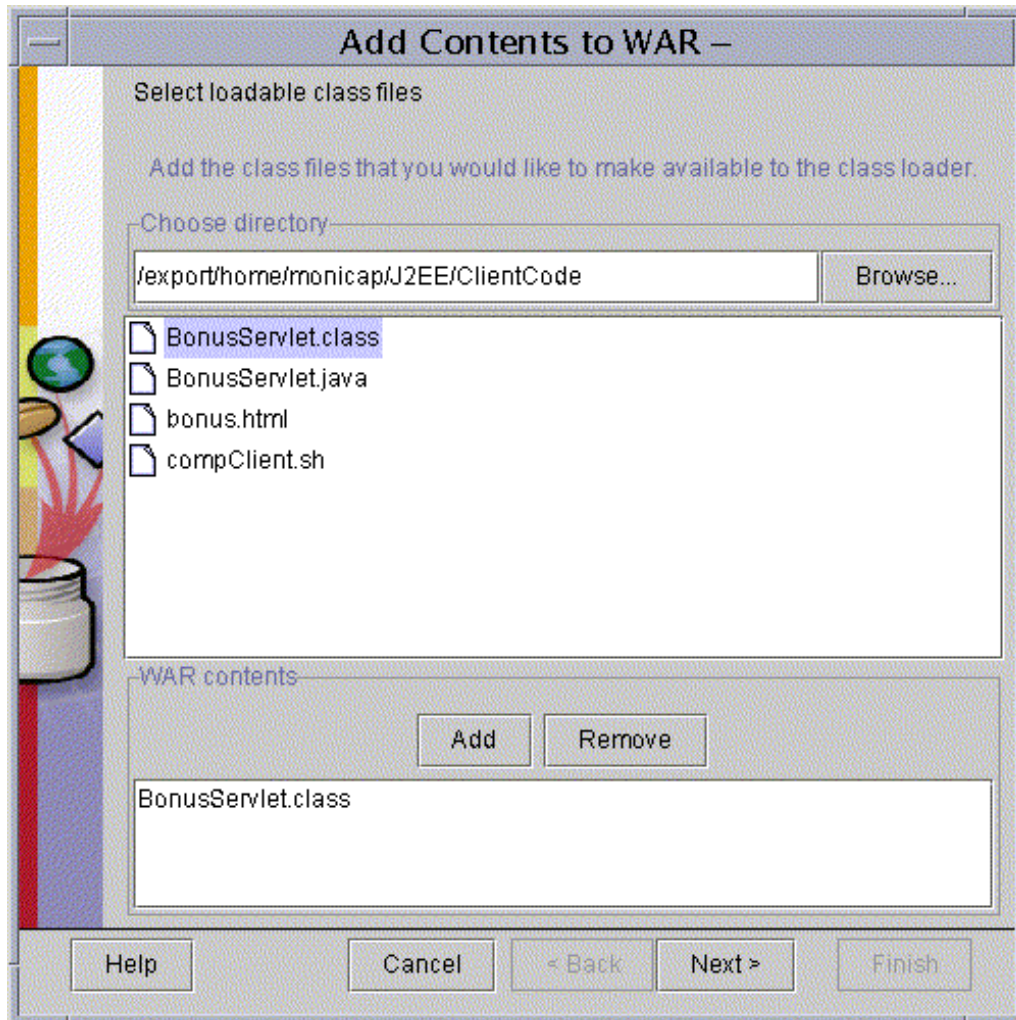


Figure 7 Add BonusServlet.class

- Click Next.
- Choose the ClientCode directory again.
- Select BonusServlet.class. Be sure the **WAR contents** shows the listing as BonusServlet.class without the ClientCode directory prefixed to the name.
- Click Add.

Add Contents to WAR dialog box: The display should look like Figure 8.

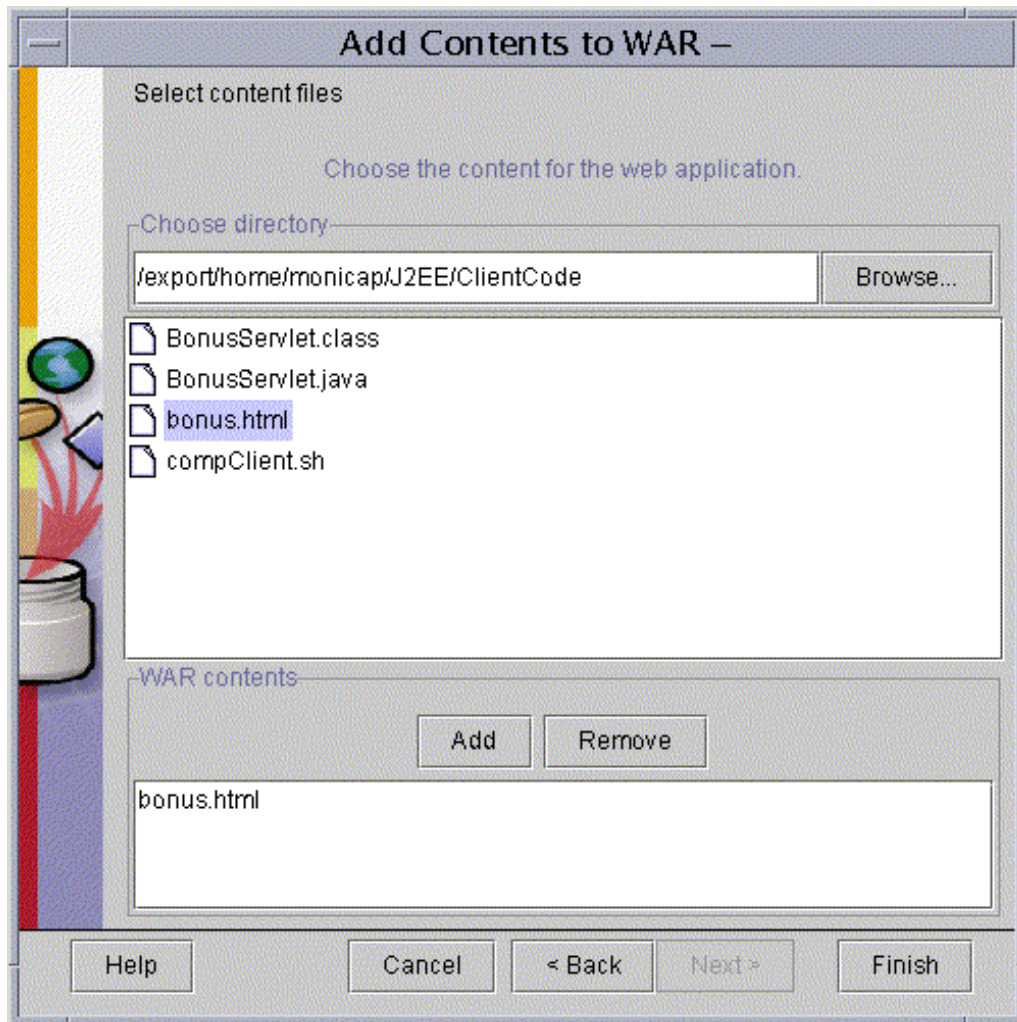


Figure 8 Add bonus.html

- Click `Finish`.

WAR File General Properties dialog box:

- Click `Next`.

Choose Component Type dialog box:

- Select **Servlet** (if it is not already selected)
- Click `Next`.

Component General Properties dialog box:

- Make sure BonusServlet is selected for the **Servlet Class**.

- Enter a display name (`BonusServlet`) and description.
- You can ignore the `Startup` and `load` sequence settings here because this example uses only one servlet.

Component Initialization Parameters dialog box:

- Click `Next`. `BonusServlet` does not use any initialization parameters.

Component Aliases dialog box:

- Click `Add`.
- Type `BonusAlias` and press `Return`. This is the same alias name you put in the `ACTION` field of the HTML form embedded in the `bonus.html` file.
- Click `Finish`.

In the Content pane, you can see that the WAR file contains an XML file with structural and attribute information on the web application, the `bonus.html` file, and the `BonusServlet` class file. The WAR file format is such that all servlet classes go in an entry starting with `Web-INF/classes`. However, when the WAR file is deployed, the `BonusServlet` class is placed in a Context Root directory under `public_html`. This placement is the convention for Servlet 2.2 compliant web servers.

To change the display name or description:

- Put your cursor in the appropriate field in the window
- Change them as you wish.
- Press the `Return` key for the edits to take effect.

Specify JNDI Name and Root Context

Before you can deploy the `BonusApp` application and its components, you have to specify the JNDI name `BonusServlet` uses to look up the `CalcBean` session bean, and specify a context root directory where the deployer will put the web components.

JNDI Name:

- Select the `BonusApp` file in the Local Applications window. The Inspecting window displays tabs at the top, and one of those tabs is JNDI Names.
- Select JNDI Names. The Inspecting window shows a three-column display with one row. `CalcBean` is listed in the middle column.
- In the far right column under JNDI name, type `calcs`. This JNDI name is the same JNDI name passed to the `BonusServlet.lookup` method.
- Press the `Return` key.

Context Root:

- Click the `Web Context` tab at the top of the Inspecting window. You will see `BonusWar` in the left column.
- Type `BonusRoot` in the right column

- Press the `Return` key. During deployment the `BonusRoot` directory is created under the `public_html` directory in your `J2sdkee1.2` installation, and the `bonus.html` file and `BonusServlet` class are copied into it as shown in Figure 9.

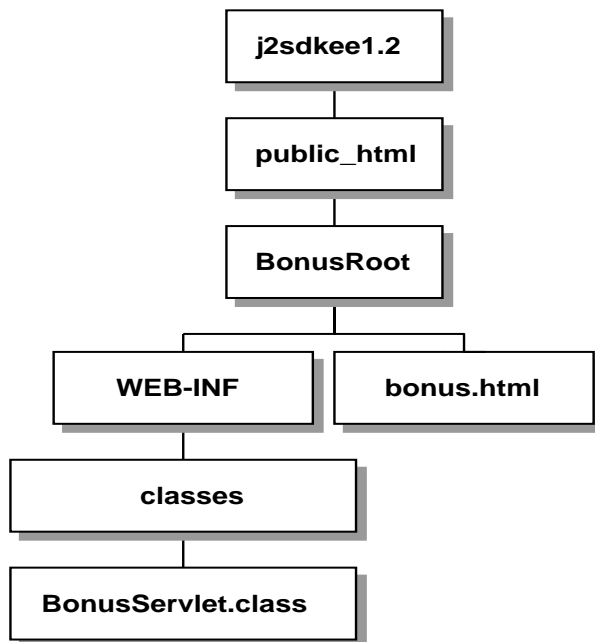


Figure 9 Context Root Directory Structure

Aliases:

- In the `LocalApp` window, click `BonusWar` and then click `BonusServlet`
- Click the `Aliases` tab at the top of the `Inspecting` window. You should see `BonusAlias` in the field.
- If `BonusAlias` is not there, type it in and press `Return`.

Verify and Deploy the J2EE Application

Before you deploy the application, it is a good idea to run the verifier. The verifier will pick up errors in the application components such as missing enterprise bean methods that the compiler does not catch.

Verify:

- With `BonusApp` selected, choose `Verifier` from the `Tools` menu.
- In the dialog that pops up, click `OK`. The window should tell you there were no failed tests.

- Close the verifier window because you are now ready to deploy the application.

Note: In the Version 1.2 software you might get a `tests app.WebURI` error. This means the deploy tool did not put a `.war` extension on the `WAR` file during `WAR` file creation. This is a minor bug and the J2EE application deploys just fine in spite of it.

Deploy:

- From the `Tools` menu, choose `Deploy Application`. A **Deploy BonusApp** dialog box pops up. Verify that the `Target Server` selection is either `localhost` or the name of the host running the J2EE server.

Note: Do not check the `Return Client Jar` box. The only time you need to check this box is when you deploy a stand-alone application for the client program. This example uses a `servlet` and `HTML` page so this box should not be checked. Checking this box creates a `JAR` file with the deployment information needed by a stand-alone application.

- Click `Next`. Make sure the `JNDI name` shows `calcs`. If it does not, type it in yourself, and press the `Return` key.
- Click `Next`. Make sure the `Context Root name` shows `BonusRoot`. If it does not, type it in yourself and press the `Return` key.
- Click `Next`.
- Click `Finish` to start the deployment. A dialog box pops up that displays the status of the deployment operation.
- When it is complete, the three bars on the left will be completely shaded as shown in Figure 10. When that happens, click `OK`.

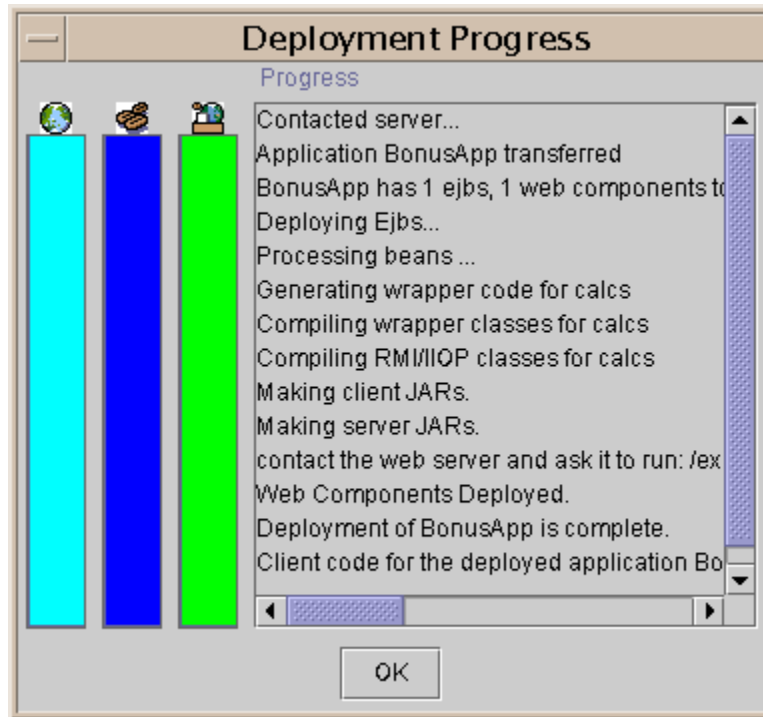


Figure 10 Deploy Application

Run the J2EE Application

The web server runs on port 8000 by default. To open the `bonus.html` page point your browser to `http://localhost:8000/BonusRoot/bonus.html`, which is where the Deploy tool put the HTML file.

Note: If you need to use a different port because port 8000 is being used for something else, edit the `web.properties` file in the `~/J2EE/j2sdkee1.2/config` directory and restart the J2EE server.

- Fill in a social security number
- Fill in a multiplier
- Click the `Submit` button. `BonusServlet` processes your data and returns an HTML page with the bonus calculation on it.

Bonus Calculation

Soc Sec: 777777777

Multiplier: 25

Bonus Amount 2500.0

Updating Component Code

The Tools menu has two menu options of interest. they are **Update Application Files** and **Update and Redeploy Application**. These options let you change code and redeploy your application with ease. Simply make your code changes, recompile the code, and choose one of these menu options.

- **Update Application Files** updates the application files with your new code. At this point you can either verify the application again or deploy it.
- **Update and Redeploy Application** updates the application files with your new code and redeploys the application without running the verifier.

Lesson 2

A Simple Entity Bean

This lesson expands the Lesson 1 example to use an entity bean. `BonusServlet` calls on the entity bean to save the social security number and bonus information to and retrieve it from a database table. This database access functionality adds the fourth and final tier to the thin-client, multitiered example started in Lesson 1.

The J2EE SDK comes with Cloudscape database, and you need no additional setup to your environment for the entity bean to access it. In fact in this example, you do not write any SQL or JDBC™ code to create the database table or perform any database access operations. The table is created and the SQL code generated with the Deploy tool during assembly and deployment. Lesson 7 JDBC Technology and Bean-Managed Persistence (page 97) shows you how to write the SQL code for an entity bean.

- Create the Entity Bean (page 28)
- Change the Servlet (page 32)
- Compile (page 34)
- Start the Platform and Tools (page 35)
- Assemble and Deploy (page 35)
- Run the J2EE Application (page 43)

Create the Entity Bean

An entity bean represents persistent data stored in one row of a database table. When an entity bean is created, the data is written to the appropriate database table row, and if the data in an entity bean is updated, the data in the appropriate database table row is also updated. The database table creation and row updates all occur without your writing any SQL or JDBC code.

Entity bean data is persistent because it survives crashes.

- If a crash occurs while the data in an entity bean is being updated, the entity bean data is automatically restored to the state of the last committed database transaction.
- If the crash occurs in the middle of a database transaction, the transaction is backed out to prevent a partial commit from corrupting the data.

BonusHome

The main difference between the `CalcHome` session bean code from Lesson 1 and the `BonusHome` entity bean code for this lesson (below) is the `findByPrimaryKey` method. This finder method takes the primary key as a parameter. In this example, the primary key is a social security number, which is used to retrieve the table row with a primary key value that corresponds to the social security number passed to this method.

The `create` method takes the bonus value and primary key as parameters. When `BonusServlet` instantiates the home interface and calls its `create` method, the container creates a `BonusBean` instance and calls its `ejbCreate` method. The `BonusHome.create` and `BonusBean.ejbCreate` methods must have the same signatures, so the bonus and primary key values can be passed from the home interface to the entity bean by way of the entity bean's container. If a row for a given primary key (social security) number already exists, a `java.rmi.RemoteException` is thrown that is handled in the `BonusServlet` client code.

```
package Beans;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import javax.ejb.EJBHome;

public interface BonusHome extends EJBHome {
    public Bonus create(double bonus, String socsec)
        throws CreateException, RemoteException;
    public Bonus findByPrimaryKey(String socsec)
        throws FinderException, RemoteException;
}
```

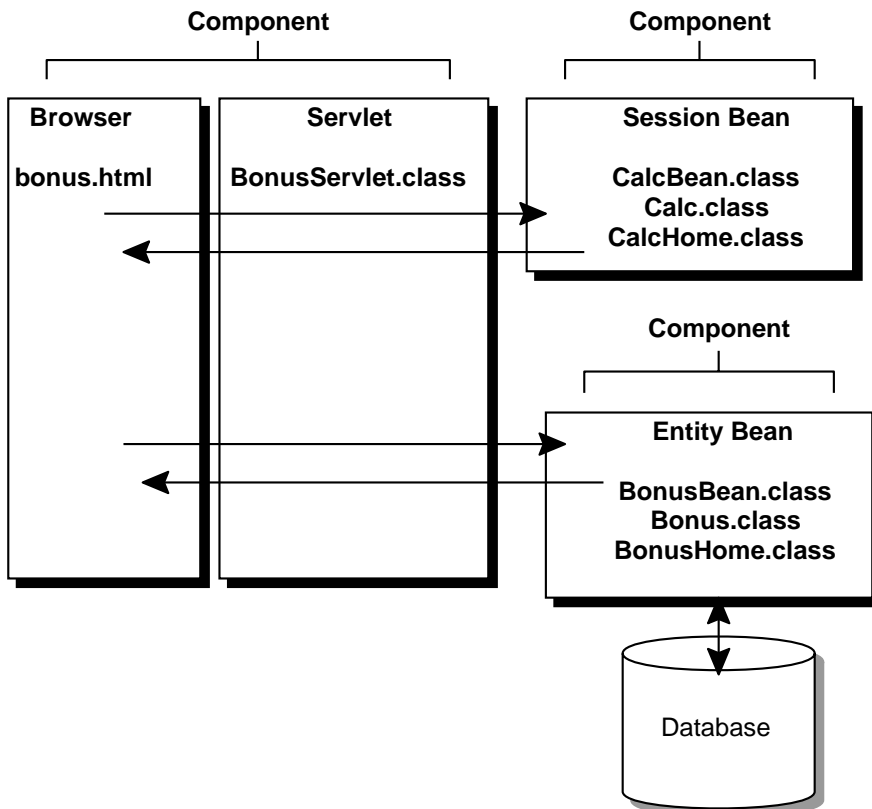
Bonus

After the home interface is created, the container creates the remote interface and entity bean. The `Bonus` interface declares the `getBonus` and `getSocSec` methods so the servlet can retrieve data from the entity bean.

```
package Beans;

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

public interface Bonus extends EJBObject {
    public double getBonus() throws RemoteException;
    public String getSocSec() throws RemoteException;
}
```



BonusBean

`BonusBean` is a container-managed entity bean. This means the container handles data persistence and transaction management without your writing code to transfer data between the entity bean and the database or define transaction boundaries.

If for some reason you want the entity bean to manage its own persistence or transactions, you would provide implementations for some of the empty methods shown in the `BonusBean` code below. The following references take you to documents that describe bean-managed persistence and transactions.

- Chapter 3 of the Writing Advanced Applications tutorial.
developer.java.sun.com/developer/onlineTraining/Programming/JDCBook
- Chapter 4 of the Java 2 Enterprise Edition Developer's Guide.
java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/DevGuideTOC.html

When `BonusServlet` calls `BonusHome.create`, the container calls the `BonusBean.setEntityContext` method. The `EntityContext` instance passed to the `setEntityContext` method has methods that let the bean return a reference to itself or get its primary key.

Next, the container calls the `ejbCreate` method. The `ejbCreate` method assigns data to the bean's instance variables, and then the container writes that data to the database. The `ejbPostCreate` method is called after the `ejbCreate` method and performs any processing needed after the bean is created. This simple example does no post-create processing.

The other empty methods are callback methods called by the container to notify the bean that some event is about to occur. You would provide behavior for some of these methods if you are using bean-managed persistence, and others if you need to provide bean-specific cleanup or initialization operations. These cleanup and initialization operations take place at specific times during the bean's lifecycle, and the container notifies the bean and calls the applicable method at the appropriate time. Here is a brief description of the empty methods:

- The `ejbPassivate` and `ejbActivate` methods are called by the container before the container swaps the bean in and out of storage. This process is similar to the virtual-memory concept of swapping a memory page between memory and disk.
- The container calls the `ejbRemove` method if the home interface has a corresponding `remove` method that gets called by the client.
- The `ejbLoad` and `ejbStore` methods are called by the container before the container synchronizes the bean's state with the underlying database.

The `getBonus` and `getSocSec` methods are called by clients to retrieve data stored in the instance variables. This example has no `set<type>` methods, but if it did, clients would call them to change the data in the bean's instance variables. Any changes to the instance variables result in an update to the table row in the underlying database.

```
package Beans;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

public class BonusBean implements EntityBean {

    public double bonus;
    public String socsec;
    private EntityContext ctx;

    public double getBonus() {
        return this.bonus;
    }
    public String getSocSec() {
        return this.socsec;
    }
}

public String ejbCreate(double bonus,
    String socsec)
    throws CreateException{
//Called by container after setEntityContext
    this.socsec=socsec;
    this.bonus=bonus;
    return null;
}

public void ejbPostCreate(double bonus,
    String socsec) {
//Called by container after ejbCreate
}

//These next methods are callback methods that
//are called by the container to notify the
//Bean some event is about to occur

public void ejbActivate() {
//Called by container before Bean
//swapped into memory
}

public void ejbPassivate() {
//Called by container before
//Bean swapped into storage
}
```

```

public void ejbRemove() throws RemoteException {
    //Called by container before
    //data removed from database
}

public void ejbLoad() {
    //Called by container to
    //refresh entity Bean's state
}

public void ejbStore() {
    //Called by container to save
    //Bean's state to database
}

public void setEntityContext(EntityContext ctx){
    //Called by container to set Bean context
}

public void unsetEntityContext(){
    //Called by container to unset Bean context
}
}

```

Change the Servlet

The BonusServlet code for this lesson is very similar to the Lesson 1 version with changes in the `init` and `doGet` methods. The `init` method for this lesson looks up both the `CalcBean` session bean, and the `BonusBean` entity bean.

```

public class BonusServlet extends HttpServlet {
    CalcHome homecalc;
    BonusHome homebonus;
    Bonus theBonus, record;

    public void init(ServletConfig config)
        throws ServletException{
        try {
            InitialContext ctx = new InitialContext();
            Object objref = ctx.lookup("bonus");
            Object objref2 = ctx.lookup("calcs");
            homebonus=(
                BonusHome)PortableRemoteObject.narrow(
                objref, BonusHome.class);
            homecalc=(CalcHome)
                PortableRemoteObject.narrow(
                objref2, CalcHome.class);

```



```

    } catch (Exception NamingException) {
        NamingException.printStackTrace();
    }
}

```

The `try` statement in the `doGet` method creates the `CalcBean` and `BonusBean` home interfaces. After calling `calcBonus` to calculate the bonus, the `BonusHome.create` method is called to create an entity bean instance and a corresponding row in the underlying database table. After creating the table, the `BonusHome.findByPrimaryKey` method is called to retrieve the same record by its primary key (social security number). Next, an HTML page is returned to the browser showing the data originally passed in, the calculated bonus, and the data retrieved from the database table row.

The `catch` statement catches and handles duplicate primary key values (social security numbers). The underlying database table cannot have two rows with the same primary key, so if you pass in the same social security number, the servlet catches and handles the error before trying to create the entity bean. In the event of a duplicate key, the servlet returns an HTML page with the original data passed in, the calculated bonus, and a duplicate key error message.

```

try {
    Calc theCalculation;
//Retrieve Bonus and Social Security Information
    String strMult = request.getParameter(
        "MULTIPLIER");//Calculate bonus
    Integer integerMult = new Integer(strMult);
    multiplier = integerMult.intValue();
    socsec = request.getParameter("SOCSEC");
//Calculate bonus
    double bonus = 100.00;
    theCalculation = homecalc.create();
    calc = theCalculation.calcBonus(
        multiplier, bonus);
//Create row in table
    theBonus = homebonus.create(calc, socsec);
    record = homebonus.findByPrimaryKey(socsec);
//Display data
    out.println("<H1>Bonus Calculation</H1>");
    out.println("<P>Soc Sec passed in: " +
        theBonus.getSocSec() + "<P>");
    out.println("<P>Multiplier passed in: " +
        multiplier + "<P>");
    out.println("<P>Bonus Amount calculated: " +
        theBonus.getBonus() + "<P>");
    out.println("<P>Soc Sec retrieved: " +
        record.getSocSec() + "<P>");
    out.println("<P>Bonus Amount retrieved: " +

```

```

        record.getBonus() + "<P>");
    out.println("</BODY></HTML>");
//Catch duplicate key error
    } catch (javax.ejb.DuplicateKeyException e) {
        String message = e.getMessage();
//Display data
    out.println("<H1>Bonus Calculation</H1>");
    out.println("<P>Soc Sec passed in: " +
        socsec + "<P>");
    out.println("<P>Multiplier passed in: " +
        multiplier + "<P>");
    out.println("<P>Bonus Amount calculated: " +
        calc + "<P>");
    out.println("<P>" + message + "<P>");
    out.println("</BODY></HTML>");
    } catch (Exception CreateException) {
        CreateException.printStackTrace();
    }
}
}

```

Compile

First, compile the entity bean and servlet. Refer to Lesson 1 for path and classpath settings, and information on where to place the source files.

Compile the Entity Bean

Unix

```

#!/bin/sh
cd /home/monicap/J2EE
J2EE_HOME=/home/monicap/J2EE/j2sdkee1.2.1
CPATH=.:$J2EE_HOME/lib/j2ee.jar
javac -d . -classpath "$CPATH" Beans/BonusBean.java
Beans/BonusHome.java Beans/Bonus.java

```

Windows

```

cd \home\monicap\J2EE
set J2EE_HOME=\home\monicap\J2EE\j2sdkee1.2.1
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar
javac -d . -classpath %CPATH% Beans/BonusBean.java
Beans/BonusHome.java Beans/Bonus.java

```

Compile the Servlet

Unix:

```
cd /home/monicap/J2EE/ClientCode
J2EE_HOME=/home/monicap/J2EE/j2sdkee1.2.1
CPATH=.:$J2EE_HOME/lib/j2ee.jar:/home/monicap/J2EE
javac -d . -classpath "$CPATH" BonusServlet.java
```

Windows:

```
cd \home\monicap\J2EE\ClientCode
set J2EE_HOME=\home\monicap\J2EE\j2sdkee1.2.1
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar;
    \home\monicap\J2EE
javac -d . -classpath %CPATH% BonusServlet.java
```

Start the Platform and Tools

To run this example, you need to start the J2EE server, the Deploy tool, and Cloudscape database. In different windows, type the following commands:

```
j2ee -verbose
deploytool
cloudscape -start
```

If that does not work, type this from the J2EE directory:

Unix

```
j2sdkee1.2.1/bin/j2ee -verbose
j2sdkee1.2.1/bin/deploytool
j2sdkee1.2.1/bin/cloudscape -start
```

Windows

```
j2sdkee1.2.1\bin\j2ee -verbose
j2sdkee1.2.1\bin\deploytool
j2sdkee1.2.1\bin\cloudscape -start
```

Assemble and Deploy

The steps in this section are:

- Update Application File
- Create Entity Bean

Update Application File

The web archive (WAR) file contains `BonusServlet` and `bonus.html`. Because you have changed `BonusServlet`, you have to update the J2EE application with the new servlet code.

- **Local Applicatons Window:** Highlight the `BonusApp` application.
- **Tools Menu:** Select **Update Application Files**.

Note: The `BonusApp` application from the previous lesson is automatically uninstalled

Create Entity Bean

The steps to creating the EJB JAR for the entity bean are very similar to the steps for the session bean covered in Lesson 1. There are a few differences, however, and those differences are explained here.

Note: In this lesson, the entity bean goes in a separate JAR file from the session bean to continue the example from Lesson 1 with the least number of changes. Because these beans have related functionality, however, you could bundle and deploy them in the same JAR file. You will see how to bundle related beans in the same JAR file in Lesson 3.

File Menu:

- Select **New Enterprise Bean**.

Introduction:

- Read and click `Next`.

EJB JAR:

- Make sure `BonusApp` shows in the **Enterprise Bean will go in field**.
- Specify `BonusJar` as the display name.
- Click `Add` (the one next to the **Contents** window).

Add Contents to JAR:

- Toggle the directory so the beans directory displays with its contents.
- Select `Bonus.class`
- Click `Add`.

- Select BonusBean.class
- Click Add.
- Select BonusHome.class
- Click Add.
- Click OK.

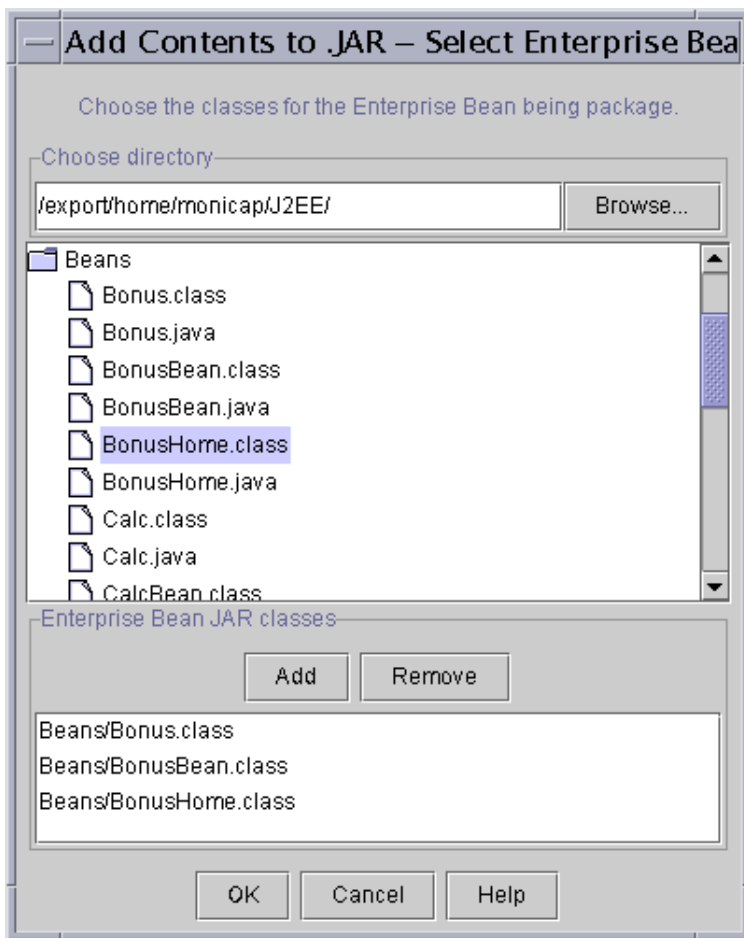


Figure 11 Adding Classes to BonusJar

EJB JAR:

- Click Next.

General:

- `Beans.BonusBean` is the classname
- `Beans.BonusHome` is the `Home` interface
- `Beans.Bonus` is the `Remote` interface.
- Enter `BonusBean` as the display name.
- Click **Entity**.
- Click `Next`.

Entity Settings:

- Select `Container-Managed persistence`.
- In the bottom window, check `bonus` and `socsec`.
- Specify `java.lang.String` for the primary key class. Note that the primary key has to be a class type. Primitive types are not valid for primary keys.
- Specify `socsec` for the primary key field name.
- Click `Next`.

Environment Entries:

- Click `Next`. This simple entity bean does not use properties (environment entries).

Enterprise Bean References:

- Click `Next`. This simple entity bean does not reference other enterprise beans.

Resource References:

- Click `Next`. This simple entity bean does not look up a database or `JavaMail™` session object.

Security:

- Click `Next`. This simple entity bean does not use security roles.

Transaction Management:

- Select `Container-managed transactions` (if it is not already selected).
- In the list below make `create`, `findByPrimaryKey`, `getBonus` and `getSocSec` required. This means the container starts a new transaction before running these methods. The transaction commits just before the methods end. There is more information on these transaction settings in *Enterprise JavaBeans Developer's Guide, Chapter 6* (java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/DevGuideTOC.html).

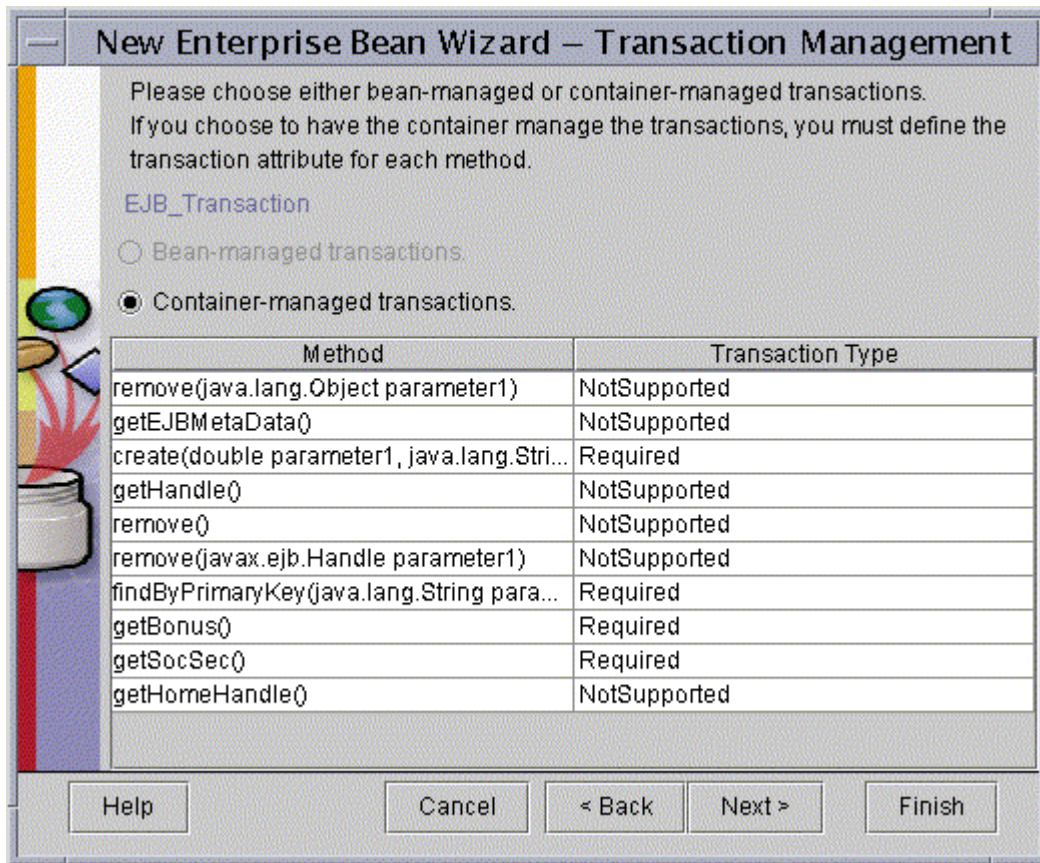


Figure 12 Transaction Management

- Click Next.
- Click Finish.

Local Applications:

- Select BonusApp.
- In the Inspecting window, select JNDI names
- Give BonusBean the JNDI name of bonus
- Press the Return key

Before the J2EE application can be deployed, you need to specify deployment settings for the entity bean and generate the SQL. Here is how to do it:

Local Applications window:

- Select BonusBean.

Inspecting window:

- Select Entity
- Click the Deployment Settings button to the lower right.

Deployment Settings:

- Specify jdbc/Cloudscape (with a capital C on Cloudscape) for the Database JNDI name
- Press Return
- Make sure the Create table on deploy and Delete table on Deploy boxes are checked.
- Click Generate SQL now.

Note: If you get an error that the connection was refused, start the database as described in Start the Platform and Tools (page 35).

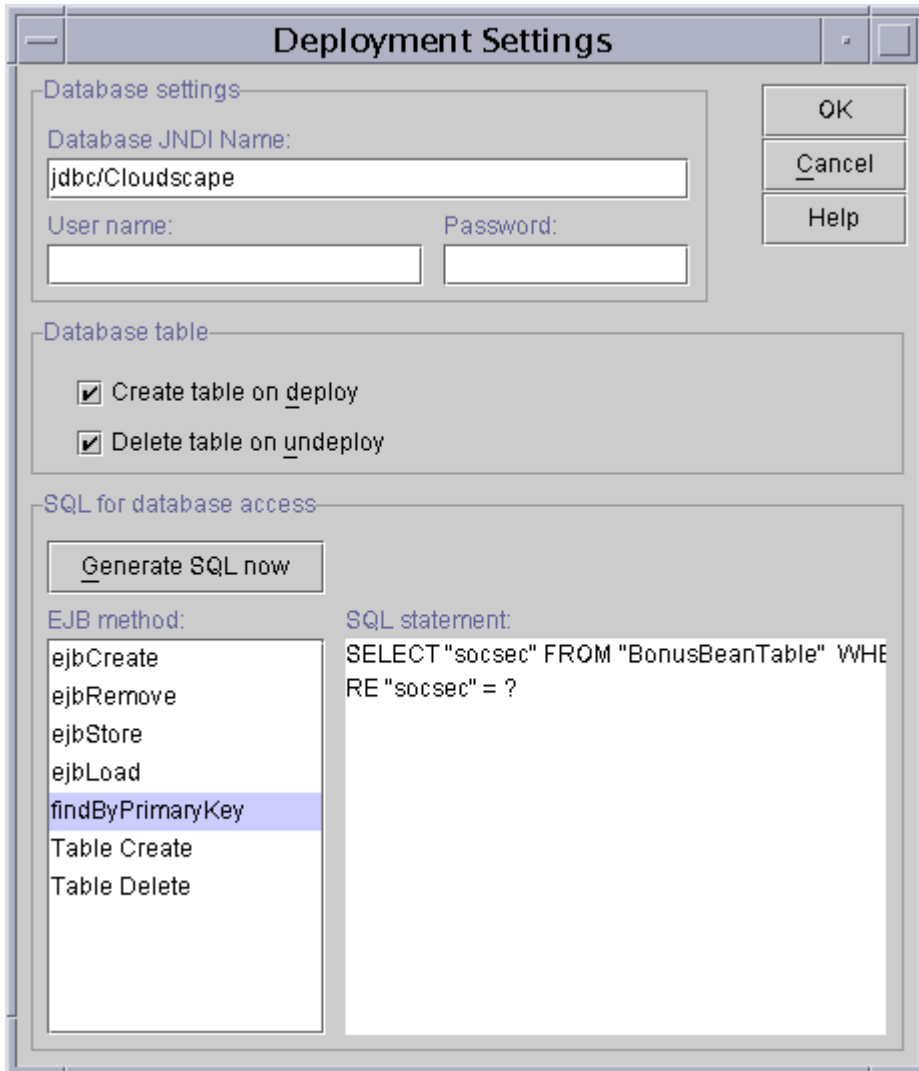


Figure 13 Generate SQL and Database Table

- When the SQL generation completes, select the `findByPrimaryKey` method in the EJB method box. To the right a SQL statement appears. It should read `SELECT "socsec" FROM "BonusBeanTable" WHERE "socsec"=?`. The question mark (?) represents the parameter passed to the `findByPrimaryKey` method.
- Click OK.

Verify and Deploy the J2EE Application

Verify:

- With BonusApp selected, choose `Verifier` from the `Tools` menu.
- In the dialog that pops up, click `OK`. The window should tell you that no tests failed.
- Close the verifier window because you are now ready to deploy the application.

Note: In the Version 1.2 software you might get a `tests app.WebURI` error. The J2EE application deploys in spite of it.

Deploy:

- **Tools Menu:** Select `Tools.Deploy Application`.

Note: Do not check the Return Client Jar box. The only time you need to check this box is when you use bean-managed persistence or deploy a stand-alone application for the client program. This example uses a servlet and HTML page so this book should not be checked. Checking this box creates a JAR file with deployment information needed by a stand-alone application.

- Click `Next`. Make sure the JNDI names show `calcs` for `CalcBean` and `bonus` for `BonusBean`. Type any missing JNDI names in yourself, and press the `Return` key.
- Click `Next`. Make sure the Context Root name shows `BonusRoot`. If it does not, type it in yourself and press the `Return` key.
- Click `Next`.
- Click `Finish` to start the deployment.
- When deployment completes, click `OK`.

Run the J2EE Application

The web server runs on port 8000 by default. To open the `bonus.html` page point your browser to `http://localhost:8000/BonusRoot/bonus.html`, which is where the Deploy tool put the HTML file.

Fill in a social security number and multiplier, and click the `Submit` button. `BonusServlet` processes your data and returns an HTML page with the bonus calculation on it.

Bonus Calculation

```
Soc Sec passed in: 777777777
Multiplier passed in: 25
Bonus Amount calculated: 2500.0
Soc Sec retrieved: 777777777
Bonus Amount retrieved: 2500.0
```

If you go back to `bonus.html` and change the multiplier to 2, but use the same social security number, you see this:

```
Bonus Calculation
Soc Sec passed in: 777777777
Multiplier passed in: 2
Bonus Amount calculated: 200.0
Duplicate primary key.
```


Lesson 3

Cooperating Enterprise Beans

In Lesson 2 A Simple Entity Bean (page 27), the servlet looks up and creates a session bean to perform a bonus calculation, and then looks up and creates an entity bean to store the bonus value and related social security number. This lesson modifies the example so the session bean looks up and creates the entity bean. Because the session and entity bean work together, they are bundled into one JAR file for deployment.

- Change the Session Bean (page 46)
- Change the Servlet (page 49)
- Compile (page 50)
- Start the Platform and Tools (page 51)
- Assemble the Application (page 52)
- Verify and Deploy the J2EE Application (page 58)
- Run the J2EE Application (page 60)

Note: Some people have trouble getting this lesson to work with 2 beans in one JAR file. If this happens to you, delete the JAR file with the two beans and put each bean in its own JAR file. You might need to stop and restart the server and tools before you can generate SQL and deploy.

Change the Session Bean

In this lesson and as shown in Figure 14, the entity bean is a client of the session bean. This means the entity bean gets its data from the session bean instead of from `BonusServlet` as it did in Lesson 2 A Simple Entity Bean (page 27). So, the `calcBonus` method in the session bean is modified to take the social security number as a parameter and create the entity bean.

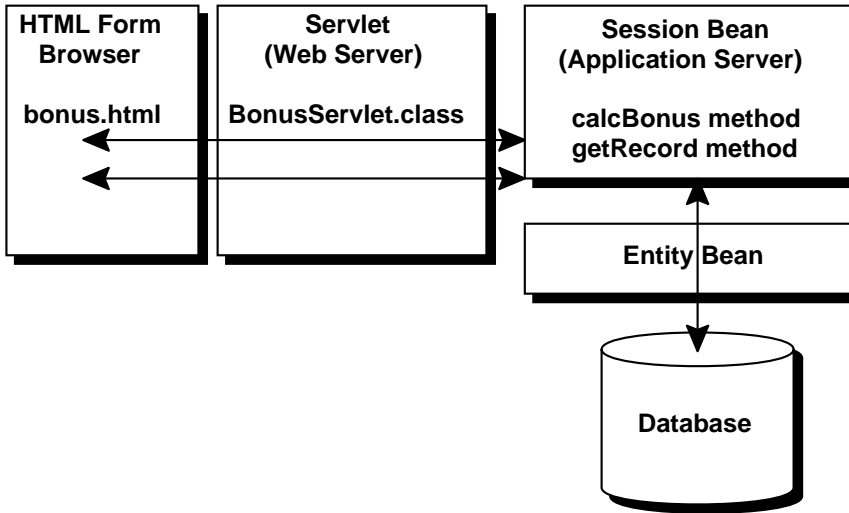


Figure 14 Beans Working Together

CalcHome

The `CalcHome` interface is unchanged. It has the same `create` method that returns an instance of the remote interface.

```

package Beans;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface CalcHome extends EJBHome {
    public Calc create()
        throws CreateException, RemoteException;
}
  
```

Calc

The `calcBonus` method in the `Calc` interface is changed to take the social security number as a parameter. This is so `CalcBean` can pass the bonus and social security number to the entity bean after calculating the bonus value. A new `getRecord` method is added so `CalcBean` can find an entity bean by its primary key (the social security number).

Also, the `calcBonus` method signature throws `DuplicateKeyException` and `CreateException`. This is so `BonusServlet` can catch and handle either of these exception conditions. `DuplicateKeyException` descends from `CreateException`. If you design the `calcBonus` method to throw `DuplicateKeyException`, but catch `CreateException`, `DuplicateKeyException` is not thrown. The way around this is to have `calcBonus` throw both `DuplicateKeyException` and `CreateException`.

```
package Beans;

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

public interface Calc extends EJBObject {
    public Bonus calcBonus(int multiplier,
                          double bonus,
                          String socsec)
        throws RemoteException,
           DuplicateKeyException,
           CreateException;
    public Bonus getRecord(String socsec)
        throws RemoteException;
}
```

CalcBean

The code to create the entity bean is moved from `BonusServlet` to the `calcBonus` method so the bonus and social security number can be written to the entity bean after the bonus is calculated. The `homebonus` variable is an instance variable so it can be used in the `calcBonus` method to look up the entity bean and in the `getRecord` method to locate the entity bean corresponding to the social security number.

```
package Beans;

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

```

import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import javax.ejb.DuplicateKeyException;
import javax.ejb.CreateException;

public class CalcBean implements SessionBean {
    BonusHome homebonus;
    //Throw DuplicateKeyException and CreateException
    //so BonusServlet can catch and handle these
    //exception conditions.
    public Bonus calcBonus(int multiplier,
                           double bonus, String socsec)
        throws DuplicateKeyException,
        CreateException {
        Bonus theBonus = null;
        double calc = (multiplier*bonus);
        try {
            InitialContext ctx = new InitialContext();
            Object objref = ctx.lookup("bonus");
            homebonus = (BonusHome)
                PortableRemoteObject.narrow(
                    objref, BonusHome.class);
        } catch (Exception NamingException) {
            NamingException.printStackTrace();
        }
        //Store data in entity bean
        try {
            theBonus = homebonus.create(calc, socsec);
        } catch (java.rmi.RemoteException e) {
            String message = e.getMessage();
            e.printStackTrace();
        }
        return theBonus;
    }

    public Bonus getRecord(String socsec) {
        Bonus record = null;
        //Use primary key to retrieve data from entity bean
        try {
            record = homebonus.findByPrimaryKey(socsec);
        } catch (java.rmi.RemoteException e) {
            String message = e.getMessage();
        } catch (javax.ejb.FinderException e) {
            e.printStackTrace();
        }
        return record;
    }
    public void ejbCreate() { }
}

```



```

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

```

Change the Servlet

The BonusServlet program is very similar to the version in Lesson 2 A Simple Entity Bean (page 27) with changes in the `init` and `doGet` methods. The `init` method for this lesson looks up the `CalcBean` session bean only.

```

public class BonusServlet extends HttpServlet {
    CalcHome homecalc;
    //Need Bonus variables because CalcBean methods
    //called in the doGet method return instances
    //of type Bonus
    Bonus theBonus, record;

    public void init(ServletConfig config)
        throws ServletException{
        try {
            InitialContext ctx = new InitialContext();
            Object objref = ctx.lookup("calcs");
            homecalc = (CalcHome)
                PortableRemoteObject.narrow(
                    objref, CalcHome.class);
        } catch (Exception NamingException) {
            NamingException.printStackTrace();
        }
    }
}

```

The `try` statement in the `doGet` method calculates the bonus, creates the session bean home interface, and calls the `calcBonus` and `getRecord` methods. If the methods successfully complete, an HTML page is returned showing the data retrieved from the entity bean. If `DuplicateKeyException` is thrown by the `calcBonus` method, an HTML page is returned showing the social security number and multiplier passed in, and the exception message, `Duplicate primary key`.

As before in Lesson 2 A Simple Entity Bean (page 27), the `catch` statement catches and handles duplicate primary key values (social security numbers).

```

    try {
        Calc theCalculation;
//Retrieve Bonus and Social Security Information
        String strMult = request.getParameter(
            "MULTIPLIER");//Calculate bonus
        Integer integerMult = new Integer(strMult);
        multiplier = integerMult.intValue();
        socsec = request.getParameter("SOCSEC");
//Calculate bonus
        double bonus = 100.00;
        theCalculation = homecalc.create();
//Call session bean
//Pass 3 parameters:multiplier, bonus, and socsec
        theBonus = theCalculation.calcBonus(
            multiplier, bonus, socsec);
        record = theCalculation.getRecord(socsec);
//Display data returned by session bean
        out.println("<H1>Bonus Calculation</H1>");
        out.println("<P>Soc Sec retrieved: " +
            record.getSocSec() + "<P>");
        out.println("<P>Bonus Amount retrieved: " +
            record.getBonus() + "<P>");
        out.println("</BODY></HTML>");
    } catch (javax.ejb.DuplicateKeyException e) {
        String message = e.getMessage();
        out.println("<H1>Bonus Calculation</H1>");
        out.println("<P>Soc Sec passed in: " + socsec +
            "<P>");
        out.println("<P>Multiplier passed in: " +
            multiplier + "<P>");
        out.println("</BODY></HTML>");
    } catch (Exception CreateException) {
        CreateException.printStackTrace();
    }
}

```

Compile

First, compile the session bean and servlet. Refer to Lesson 1 for path and classpath settings, and information on where to place the source files.

Compile the Session Bean

Unix

```
#!/bin/sh
cd /home/monicap/J2EE
J2EE_HOME=/home/monicap/J2EE/j2sdkee1.2.1
CPATH=.:$J2EE_HOME/lib/j2ee.jar
javac -d . -classpath "$CPATH" Beans/CalcBean.java
        Beans/CalcHome.java Beans/Calc.java
```

Windows

```
cd \home\monicap\J2EE
set J2EE_HOME=\home\monicap\J2EE\j2sdkee1.2.1
set CPATH=.;%J2EE_HOME%\lib\j2ee.jar
javac -d . -classpath %CPATH% Beans/CalcBean.java
        Beans/CalcHome.java Beans/Calc.java
```

Compile the Servlet

Unix:

```
cd /home/monicap/J2EE/ClientCode
J2EE_HOME=/home/monicap/J2EE/j2sdkee1.2
CPATH=.:$J2EE_HOME/lib/j2ee.jar:
        /home/monicap/J2EE
javac -d . -classpath "$CPATH" BonusServlet.java
```

Windows:

```
cd \home\monicap\J2EE\ClientCode
set J2EE_HOME=\home\monicap\J2EE\j2sdkee1.2 set
CPATH=.;%J2EE_HOME%\lib\j2ee.jar:\home\monicap\J2EE
javac -d . -classpath %CPATH% BonusServlet.java
```

Start the Platform and Tools

To run this example, you need to start the J2EE server, the Deploy tool, and Cloudscape database. In different windows, type the following commands:

```
j2ee -verbose
deploytool
cloudscape -start
```

If that does not work, type this from the J2EE directory:

Unix

```
j2sdkee1.2.1/bin/j2ee -verbose  
j2sdkee1.2.1/bin/deploytool  
j2sdkee1.2.1/bin/cloudscape -start
```

Windows

```
j2sdkee1.2.1\bin\j2ee -verbose  
j2sdkee1.2.1\bin\deploytool  
j2sdkee1.2.1\bin\cloudscape -start
```

Assemble the Application

The steps for this section include the following:

- Create New J2EE Application
- Create New Web Component
- Bundle Session and Entity Beans in One JAR File

Create New J2EE Application

Rather than update the J2EE application from Lessons 1 and 2, these steps create a new J2EE application.

Delete `BonusApp`:

- Click `BonusApp` so it is highlighted
- Select **Delete** from the **Edit** menu

Create `2BeansApp`:

- From the **File** menu, select **New Application**.
- Click the right mouse button in the **Application Display Name** field. `2BeansApp` appears as the display name.
- Click the **Browse** button to open the file chooser to select the location where you want the application `EAR` file to be saved.

New Application file chooser:

- Locate the directory where you want to place the application `EAR` file
- In this example, that directory is `/export/home/monicap/J2EE`.
- In the **File name** field, type `2BeansApp.ear`.

- Click **New Application**.
- Click **OK**.

Create New Web Component

Now, go through the steps to create the WAR file. These steps are outlined in Lesson 1 and summarized below. With 2BeansApp selected,

File Menu:

- Select `New Web Component`.

Introduction:

- Read and Click `Next`

War File General Properties:

- Specify `BonusWar` for the display name.
- Click `Add`
- Go to the `ClientCode` directory and add `bonus.html`
- Click `Next`
- Go to the `ClientCode` directory and add `BonusServlet.class`
- Click `Finish`.

War File General Properties:

- Click `Next`.

Choose Component Type:.

- Make sure `Describe a servlet` is selected.
- Click `Next`.

Component General Properties:

- Make `BonusServlet` the servlet class
- Make the display name `BonusServlet`.
- Click `Next`.

Component Initialization Parameters.

- Click `Next`.

Component Aliases:

- Specify `BonusAlias`
- Click `Finish`.

Inspecting window:

- Select Web Context
- Specify `BonusRoot`.

Bundle Session and Entity Beans in one JAR File

In this lesson, you will put both the session and entity beans in the same JAR file. To do this, you first create the JAR file with only the session bean in it, and then add the entity bean to that JAR file.

Create JAR with Session Bean

With `2BeansApp` selected,

File Menu:

- Select `New Enterprise Bean`

Introduction:

- Read and click `Next`.

EJB JAR:

- Make sure `2BeansApp` shows in the **Enterprise Bean will go in** field.
- Specify `2BeansJar` as the display name.
- Click `Add` (the one next to the **Contents** window).
- Toggle the directory so the Beans directory displays with its contents.
- Select `Calc.class`
- Click `Add`.
- Select `CalcBean.class`
- Click `Add`.
- Select `CalcHome.class`
- Click `Add`.

Enterprise Bean JAR classes:

- Make sure you see `Beans/Calc.class`, `Beans/CalcHome.class`, and `Beans/CalcBean.class` in the display.
- Click `OK`.

EJB JAR:

- Click `Next`.

General:

- `CalcBean` is the classname, `Beans.CalcHome` is the `Home` interface, and `Beans.Calc` is the `Remote` interface.
- Enter `CalcBean` as the display name.
- Click `session` and `stateless`.
- Click `Next`.

Environment Entries:

- Click `Next`. This simple session bean does not use properties (environment entries).

Enterprise Bean References:

- Click `Next`. The references are handled during deployment rather than here.

Resource References:

- Click `Next`. This simple session bean does not look up a database or `JavaMail™` session object.

Security:

- Click `Next`. This simple session bean does not use security roles.

Transaction Management:

- Select `Container-managed transactions` (if it is not already selected).
- In the list below make `calcBonus`, and `getRecord` required. This means the container starts a new transaction before running these methods. The transaction commits just before the methods end. You can find more information on these transaction settings in Chapter 6 of the *Enterprise JavaBeans Developer's Guide*.
- Click `Next`.

Review Settings:

- Click `Finish`.

Local Applications:

- Select `2BeansApp`.
- In the `Inspecting` window, select `JNDI names`, give `CalcBean` the `JNDI` name of `calcs`, and press the `Return` key.

Add the Entity Bean

With `2BeansApp` selected,

File Menu:

- Select `New Enterprise Bean`

Introduction:

- Read and click `Next`.

EJB JAR:

- Make sure `2BeansJar` shows in the **Enterprise Bean will go in** field. This setting will add the new bean to the existing JAR file instead of putting the new bean in its own JAR file.
- Click `Add` (the one next to the **Contents** window).
- Toggle the directory so the Beans directory displays with its contents.
- Select `Bonus.class`
- Click `Add`.
- Select `BonusBean.class`
- Click `Add`.
- Select `BonusHome.class`
- Click `Add`.

Enterprise Bean JAR classes:

- Make sure you see `Beans/Bonus.class`, `Beans/BonusHome.class`, and `Beans/BonusBean.class` in the display.
- Click `OK`.

EJB JAR:

- Click `Next`.

General:

- Make sure `Beans.BonusBean` is the classname, `Beans.BonusHome` is the Home interface, and `Beans.Bonus` is the Remote interface.
- Enter `BonusBean` as the display name.
- Click **Entity**.
- Click `Next`.

Entity Settings:

- Select `Container managed persistence`.
- In the window below, check `bonus` and `socsec`. The primary key class is `java.lang.String`, and the primary key field name is `socsec`. Note that the primary key has to be a class type. Primitive types are not valid for primary keys.
- Click `Next`.

Environment Entries:

- Click `Next`. This simple entity bean does not use properties (environment entries).

Enterprise Bean References:

- Click `Next`. This simple entity bean does not reference other enterprise Beans.

Resource References:

- Click `Next`. This simple entity bean does not look up a database or JavaMail™ session object.

Security:

- Click `Next`. This simple entity bean does not use security roles.

Transaction Management:

- Select `Container-managed transactions` (if it is not already selected).
- In the list below make `create`, `findByPrimaryKey`, `getBonus` and `getSocSec` required. This means the container starts a new transaction before running these methods. The transaction commits just before the methods end. You can find more information on these transaction settings in Chapter 6 of the Enterprise JavaBeans Developer's Guide.
- Click `Next`.

Review Settings:

- Click `Finish`.

Local Applications:

- Select `2BeansApp`.
- In the Inspecting window, select `JNDI names`, give `BonusBean` the JNDI name of `bonus` and `CalcBean` the JNDI name of `calcs`
- Press the Return key after each entry.

Before the J2EE application can be deployed, you need to specify deployment settings for the entity bean and generate the SQL. Here is how to do it:

Local Applications window:

- Select `BonusBean`.

Inspecting window:

- Select `Entity`
- Click the `Deployment Settings` button to the lower right.

Deployment Settings window:

- Specify `jdbc/Cloudscape` (with a capital *C* on Cloudscape) for the Database JNDI name
- Press Return
- Make sure the `Create table on deploy` and `Delete table on Deploy` boxes are checked
- Click `Generate SQL now`.

Note: If you get an error that the connection was refused, start the database as described in *Start the Platform and Tools* (page 51).

When the SQL generation completes,

- Select the `findByPrimaryKey` method in the EJB method box.
- To the right a SQL statement appears. It should read `SELECT "socsec" FROM "Bonus-BeanTable" WHERE "socsec"=?`. The question mark (?) represents the parameter passed to the `findByPrimaryKey` method.
- Click `OK`.

Verify and Deploy the J2EE Application

Before you deploy the application, it is a good idea to run the verifier. The verifier will pick up errors in the application components such as missing enterprise bean methods that the compiler does not catch.

Note: If you get a `Save` error when you verify or deploy, shut everything down and restart the server and tools.

Verify:

- With `2BeansApp` selected, choose `Verifier` from the `Tools` menu.
- In the dialog that pops up, click `OK`. The window should tell you there were no failed tests.
- Close the verifier window because you are now ready to deploy the application.

Note: In the Version 1.2.1 software you might get a `tests app.WebURI` error. This means the deploy tool did not put a `.war` extension on the `WAR` file during `WAR` file creation. This is a minor bug and the J2EE application deploys just fine in spite of it.

Deploy:

- From the **Tools** menu, choose `Deploy Application`. A **Deploy BonusApp** dialog box pops up.
- Verify that the `Target Server` selection is either `localhost` or the name of the host running the J2EE server.

Note: Do not check the `Return Client Jar` box. The only time you need to check this box is when you use bean-managed persistence or deploy a stand-alone application for the client program. This example uses a servlet and HTML page so this book should not be checked. Checking this box creates a JAR file with deployment information needed by a stand-alone application.

- Click `Next`.
- Make sure the JNDI names show for `calcs` for `CalcBean` and `bonus` for `BonusBean`. If they do not, type the JNDI names in yourself, and press the `Return` key.
- Click `Next`. Make sure the `Context Root` name shows `BonusRoot`. If it does not, type it in yourself and press the `Return` key.
- Click `Next`.
- Click `Finish` to start the deployment. A dialog box pops up that displays the status of the deployment operation.
- When it is complete, the three bars on the left will be completely shaded as shown in Figure 15. When that happens, click `OK`.

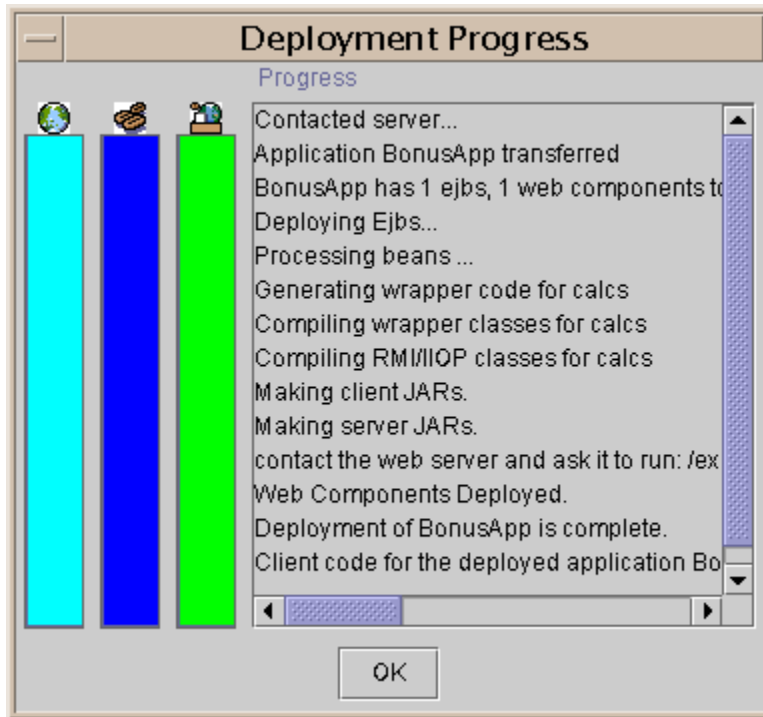


Figure 15 Deploy Application

Run the J2EE Application

The web server runs on port 8000 by default. To open the `bonus.html` page point your browser to `http://localhost:8000/BonusRoot/bonus.html`, which is where the Deploy tool put the HTML file.

- Fill in a social security number and multiplier.
- Click the `Submit` button. `BonusServlet` processes your data and returns an HTML page with the bonus calculation on it.

Bonus Calculation

```
Soc Sec retrieved: 777777777
Bonus Amount Retrieved: 200.0
```

If you supply the same social security number twice, you will see something similar to this:

Bonus Calculation

```
Soc Sec passed in: 777777777
Multiplier passed in: 2
Duplicate primary key
```

Lesson 4

JavaServer Pages Technology

JavaServer Pages™ (JSP) technology lets you put segments of servlet code directly into a static HTML page. When the JSP Page is loaded by a browser, the servlet code executes and the application server creates, compiles, loads, and runs a background servlet to execute the servlet code segments and return an HTML page or print an XML report.

This lesson changes the WAR file from Lesson 3 Cooperating Enterprise Beans (page 45) to use a JSP Page instead of `BonusServlet`.

- Create the JSP Page (page 62)
- Change `bonus.html` (page 66)
- Start the Platform and Tools (page 67)
- Remove the WAR File (page 67)
- Create New WAR File (page 67)
- Verify and Deploy the J2EE Application (page 68)
- Run the J2EE Application (page 70)
- More Information (page 71)

Create the JSP Page

A JSP Page looks like an HTML page with servlet code segments embedded between various forms of leading (<%) and closing (%>) JSP tags. There are no `HttpServlet` methods such as `init`, `doGet`, or `doPost`. Instead, the code that would normally be in these methods is embedded directly in the JSP Page using JSP scriptlet tags.

The following JSP Page (`Bonus.jsp`) is equivalent to `BonusServlet` from Lesson 3 Cooperating Enterprise Beans (page 45). A more detailed description of the JSP tags follows the code listing. Note that JSP tags cannot be nested. For example, you cannot nest a JSP comment tag within a JSP scriptlet tag.

```
<HTML>
<HEAD>
<TITLE>Bonus Calculation</TITLE>
</HEAD>
  <!-- Comment
    Scriptlet for import statements
    <% indicates a jsp directive --%>
  <%@ page import="javax.naming.*" %>
  <%@ page import="javax.rmi.PortableRemoteObject" %>
  <%@ page import="Beans.*" %>
  <!-- Comment
    Scriptlet to get the parameters,
    convert string to Integer to int for bonus
    calculation, and declare/initialize bonus
    variable. <% indicates a jsp scriptlet --%>
  <%! String strMult, socsec; %>
  <%! Integer integerMult; %>
  <%! int multiplier; %>
  <%! double bonus; %>
<%
  strMult = request.getParameter("MULTIPLIER");
  socsec = request.getParameter("SOCSEC");
  integerMult = new Integer(strMult);
  multiplier = integerMult.intValue();
  bonus = 100.00;
%>
  <!-- Comment
    Scriptlet to look up session Bean --%>
<%
  InitialContext ctx = new InitialContext();
  Object objref = ctx.lookup("calcs");
  CalcHome homecalc = (CalcHome)
    PortableRemoteObject.narrow(
      objref, CalcHome.class);
%>
```

```

<%-- Comment
  Scriptlet to create session Bean,
  call calcBonus method, and retrieve a database
  record by the social security number
  (primary key) --%>
<%
try {
  Calc theCalculation = homecalc.create();
  Bonus theBonus = theCalculation.calcBonus(
                        multiplier,
                        bonus,
                        socsec);
  Bonus record = theCalculation.getRecord(socsec);
%>
<%-- Comment
  HTML code to display retrieved data
  on returned HTML page. --%>
<H1>Bonus Calculation</H1>
  Social security number retrieved:
      <%= record.getSocSec() %>
  <P>
  Bonus Amount retrieved: <%= record.getBonus() %>
  <P>
<%-- Comment
  Scriptlet to catch DuplicateKeyException --%>
<%
} catch (javax.ejb.DuplicateKeyException e) {
  String message = e.getMessage();
%>
<%-- Comment
  HTML code to display original data passed to JSP
  on returned HTML page --%>
  Social security number passed in: <%= socsec %>
  <P>
  Multiplier passed in: <%= strMult %>
  <P>
  Error: <%= message %>
<%-- Comment
  Scriptlet to close try and catch block --%>
<%
}
%>
<%-- Comment
  HTML code to close HTML body and page --%>
</BODY>
</HTML>

```

Comments

The first seven lines of `Bonus.jsp` show straight HTML followed by a JSP comment. JSP comments are similar to HTML comments except they start with `<%--` instead of `<!--`, which is how they look in HTML. You can use either JSP or HTML comments in a JSP file. HTML comments are sent to the client's web browser where they appear as part of the HTML page, and JSP comments are stripped out and do not appear in the generated HTML.

Note: I found that putting a colon in a JSP comment as in `<%-- Comment: Scriptlet for import statements . . .` created a runtime error that went away when I took the colon out.

```
<HTML>
<HEAD>
<TITLE>Bonus Calculation</TITLE>
</HEAD>
  <%-- Comment
    Scriptlet for import statements
    <%@ indicates a jsp directive --%>
```

Directives

JSP directives are instructions processed by the JSP engine when the JSP Page is translated to a servlet. The directives used in this example tell the JSP engine to include certain packages and classes. Directives are enclosed by the `<%@` and `%>` directive tags.

```
<%@ page import="javax.naming.*" %>
<%@ page import="javax.rmi.PortableRemoteObject" %>
<%@ page import="Beans.*" %>
```

Declarations

JSP declarations let you set up variables for later use in expressions or scriptlets. You can also declare variables within expressions or scriptlets at the time you use them. The scope is the entire JSP Page, so there is no concept of instance variables. That is, you do not have to declare instance variables to be used in more than one expression or scriptlet. Declarations are enclosed by the `<%!` and `%>` declaration tags. You can have multiple declarations. For example, `<%! double bonus; String text; %>`.

```
<%! String strMult, socsec; %>
<%! Integer integerMult; %>
<%! int multiplier; %>
<%! double bonus; %>
```


Scriptlets

JSP scriptlets let you embed java code segments into the JSP page. The embedded code is inserted directly into the generated servlet that executes when the page is requested. This scriptlet uses the variables declared in the directives described above. Scriptlets are enclosed by the `<%` and `%>` scriptlet tags.

```
<%
    strMult = request.getParameter("MULTIPLIER");
    socsec = request.getParameter("SOCSEC");
    integerMult = new Integer(strMult);
    multiplier = integerMult.intValue();
    bonus = 100.00;
%>
```

Predefined Variables

A scriptlet can use the following predefined variables: `session`, **`request`**, `response`, `out`, and `in`. This example uses the `request` predefined variable, which is an `HttpServletRequest` object. Likewise, `response` is an `HttpServletResponse` object, `out` is a `PrintWriter` object, and `in` is a `BufferedReader` object.

Predefined variables are used in scriptlets in the same way they are used in servlets, except you do not declare them.

```
<%
    strMult = request.getParameter("MULTIPLIER");
    socsec = request.getParameter("SOCSEC");
    integerMult = new Integer(strMult);
    multiplier = integerMult.intValue();
    bonus = 100.00;
%>
```

Expressions

JSP expressions let you dynamically retrieve or calculate values to be inserted directly into the JSP Page. In this example, an expression retrieves the social security number from the Bonus entity bean and puts it on the JSP page.

```
<H1>Bonus Calculation</H1>
    Social security number retrieved:
        <%= record.getSocSec() %>
    <P>
    Bonus Amount retrieved: <%= record.getBonus() %>
    <P>
```

JSP-Specific Tags

The JavaServer Pages 1.1 specification defines JSP-specific tags that let you extend the JSP implementation with new features and hide a lot of complexity from visual designers who need to look at the JSP page and modify it. The JSP example in this lesson does not use any JSP-specific tags, but you will see an example of these tags in the next lesson. The JSP-specific tags defined in the 1.1 specification are the following:

`jsp:forward` and `jsp:include` to instruct the JSP engine to switch from the current page to another JSP page.

`jsp:useBean`, `jsp:setProperty`, and `jsp:getProperty` let you embed and use JavaBeans technology inside a JSP Page.

`jsp:plugin` automatically downloads the appropriate Java Plug-In to the client to execute an applet with the correct Java platform.

Change bonus.html

The only change you need to make to `bonus.html` is to have the `ACTION` parameter in the HTML form invoke `Bonus.jsp` instead of `BonusServlet`.

```
<HTML>
<BODY BGCOLOR = "WHITE">
<BLOCKQUOTE>
<H3>Bonus Calculation</H3>
<FORM METHOD="GET" ACTION="Bonus.jsp">
<P>
Enter social security Number:
<P>
<INPUT TYPE="TEXT" NAME="SOCSEC"></INPUT>
<P>
Enter Multiplier:
<P>
<INPUT TYPE="TEXT" NAME="MULTIPLIER"></INPUT>
<P>
<INPUT TYPE="SUBMIT" VALUE="Submit">
<INPUT TYPE="RESET">
</FORM>
</FORM>
</BLOCKQUOTE>
</BODY>
</HTML>
```

Start the Platform and Tools

To run this example, you need to start the J2EE server, the Deploy tool, and Cloudscape database. In different windows, type the following commands:

```
j2ee -verbose
deploytool
cloudscape -start
```

If that does not work, type this from the `J2EE` directory:

Unix

```
j2sdkee1.2.1/bin/j2ee -verbose
j2sdkee1.2.1/bin/deploytool
j2sdkee1.2.1/bin/cloudscape -start
```

Windows

```
j2sdkee1.2.1\bin\j2ee -verbose
j2sdkee1.2.1\bin\deploytool
j2sdkee1.2.1\bin\cloudscape -start
```

Remove the WAR File

Because a JSP page is added to the Web component, you have to delete the WAR file from the previous lesson and create a new one with the JSP page in it.

Local Applications:

- Click the `2BeansApp` icon so you can see its application components.
- Select `BonusWar` so it is outlined and highlighted.
- Select `Delete` from the **Edit** menu.

Create New WAR File

File menu:

- Select `New Web Component`

Introduction:

- Read and Click `Next`.

War File General Properties:

Note: There appears to be a bug in the Deploy tool. Make sure you add `Bonus.jsp` first followed by `bonus.html`. If you add `bonus.html` first, Deploy tool puts `bonus.html` where `Bonus.jsp` should go, and `Bonus.jsp` where `bonus.html` should go. If this happens, you can manually fix the problem by copying them to their correct locations. This is where they correctly belong after deployment:

```
~/j2sdkee1.2/public_html/JSPRoot/bonus.html
```

```
~/j2sdkee1.2/public_html/JSPRoot/WEB-INF/classes/Bonus.jsp
```

- Specify `BonusWar` for the display name.
- Click `Add`
- Go to the `ClientCode` directory and add `Bonus.jsp`,
- Click `Next`
- Go to the `ClientCode` directory and add `bonus.html`
- Click `Finish`.

War File General Properties:

- Click `Next`.

Choose Component Type:

- Make sure `Describe a JSP` is selected. Click `Next`.

Component General Properties:

- Make `Bonus.jsp` the JSP filename
- Make the display name `BonusJSP`.
- Click `Finish`.

Inspecting window:

- Select `Web Context`
- Specify `JSPRoot`.

Verify and Deploy the J2EE Application

Before you deploy the application, it is a good idea to run the verifier. The verifier will pick up errors in the application components such as missing enterprise Bean methods that the compiler does not catch.

Verify:

- With `2BeansApp` selected, choose `Verifier` from the `Tools` menu.
- In the dialog that pops up, click `OK`. The window should tell you no tests failed.
- Close the verifier window because you are now ready to deploy the application.

Deploy:

- From the `Tools` menu, choose `Deploy Application`. A **Deploy BonusApp** dialog box pops up.
- Verify that the `Target Server` selection is either `localhost` or the name of the host running the J2EE server.

Note: Do not check the `Return Client Jar` box. The only time you need to check this box is when you deploy a stand-alone application for the client program. This example uses an HTML and JSP page so this book should not be checked. Checking this box creates a JAR file with deployment information needed by a stand-alone application.

- Click `Next`. Make sure the JNDI names show `calcs` for `CalcBean` and `bonus` for `BonusBean`. If they do not show these names, type them in yourself, and press the `Return` key.
- Click `Next`. Make sure the `Context Root` name shows `JSPRoot`. If it does not, type it in yourself and press the `Return` key.
- Click `Next`.
- Click `Finish` to start the deployment. A dialog box pops up that displays the status of the deployment operation.
- When it is complete, the three bars on the left will be completely shaded as shown in Figure 16. When that happens, click `OK`.

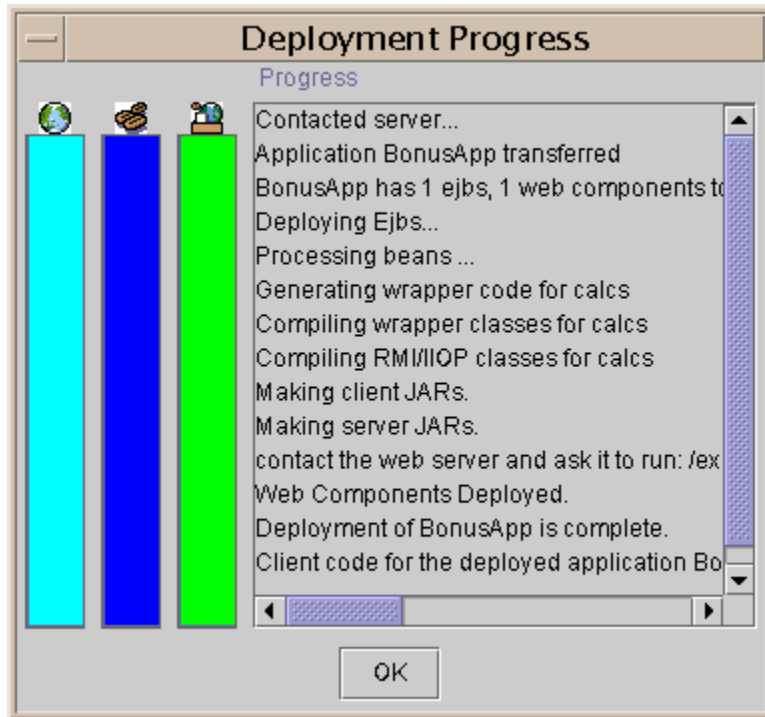


Figure 16 Deploy Application

Run the J2EE Application

The web server runs on port 8000 by default. To open the `bonus.html` page point your browser to `http://localhost:8000/JSPRoot/bonus.html`, which is where the Deploy tool put the HTML file.

Note: Deploy tool puts `Bonus.jsp` under `public_html/JSPRoot`, and `bonus.html` under `public_html/JSPRoot/WEB-INF/classes`, which is opposite of where they really belong. Manually copy them to their correct locations as follows: `public_html/JSPRoot/bonus.html` and `public_html/JSPRoot/WEB-INF/classes/Bonus.jsp`.

- Fill in a social security number and multiplier
- Click the `Submit` button. `Bonus.jsp` processes your data and returns an HTML page with the bonus calculation on it.

Bonus Calculation

```
Social Security number retrieved: 777777777  
Bonus Amount Retrieved: 200.0
```

If you supply the same social security number twice, you will see something similar to this:

Bonus Calculation

```
Soc Sec passed in: 777777777  
Multiplier passed in: 2  
Error: Duplicate primary key
```

More Information

Another way to use JavaServer pages technology is in combination with JavaBeans™ technology where the JSP page presents a form to the user and calls on the JavaBean to process the data entered on the form. You can see an example at the following URL: <http://java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Client.fm.html#10649>

This next URL takes you to an article with a great explanation of JavaServer pages and JavaBeans technologies: *Building Your own JSP Components*
http://developer.iplanet.com/viewsource/fields_jspcomp/fields_jspcomp.html

Lesson 5

Adding JavaBeans Technology to the Mix

You can use JavaBeans™ technology to put a JavaBean between the JSP page and `CalcBean` session bean to get a better Model, View, Controller (MVC) separation. MVC is a design pattern that consists of three kinds of objects. The Model provides the application business logic, the View is its screen presentation, and the Controller is an object that manages what happens when the user interacts with the View. A design pattern describes a recurring problem and its solution where the solution is never exactly the same for every recurrence.

Lesson 4 JavaServer Pages Technology (page 61) is set up so the HTML and JSP pages provide the screen presentation (View) and manage what happens when the user interacts with the data (Controller). The entity and session bean (`BonusBean` and `CalcBean`) are the application objects or Model.

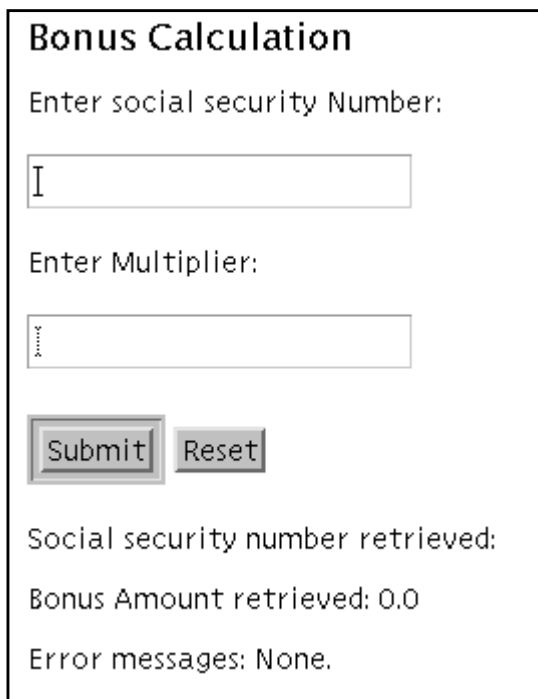
This lesson uses a JSP page for the screen presentation (View), a JavaBean to manage what happens when the user interacts with the View (Controller), and the entity and session beans for the application objects (Model). Separating the Controller from the View like this lets the JavaBean serve as a wrapper for the session bean and gives the example a much cleaner MVC separation. An application that uses clear design patterns is easier to update, maintain, and manage.

- About the Example (page 74)
- Create `bonus.jsp` (page 76)
- Create the JavaBeans Class (page 79)
- Bean Properties (page 81)
- Remove the WAR File (page 85)
- Create New WAR File (page 85)
- Verify and Deploy the J2EE Application (page 86)
- Run the J2EE Application (page 87)
- More Information (page 87)

About the Example

In Lesson 4 JavaServer Pages Technology (page 61), the application user interface consisted of an HTML page with an HTML form. The HTML form calls the JSP page when the user clicks the `Submit` button on the HTML page.

Another way to create the user interface is with one JSP page that includes the HTML form, JSP scriptlets, and JSP-specific tags for interacting with the JavaBean. When the JSP page loads, the HTML form is displayed and the scriptlet and JSP-specific tags for interacting with the JavaBean are executed. Because no data has been supplied yet, the display looks like Figure 17:



The screenshot shows a web form titled "Bonus Calculation" enclosed in a rectangular border. The form contains the following elements from top to bottom: a label "Enter social security Number:" followed by a text input field with a cursor; a label "Enter Multiplier:" followed by a text input field with a cursor; two buttons labeled "Submit" and "Reset"; and three lines of feedback text: "Social security number retrieved:", "Bonus Amount retrieved: 0.0", and "Error messages: None."

Figure 17 When `bonus.jsp` Loads

After the user enters some data and clicks the `Submit` button, the HTML form is redisplayed and the scriptlet and JSP-specific tags for interacting with the JavaBean execute again with the data supplied. The display looks something like Figure 18. This is because the `ACTION` parameter for the HTML form on `bonus.jsp` recursively calls itself.

Bonus Calculation

Enter social security Number:

Enter Multiplier:

Social security number retrieved: 777777777

Bonus Amount retrieved: 200.0

Error messages:

Figure 18 After User Enters Data and Clicks Submit

If the user enters the same social security number, a duplicate key error is returned and displayed on the JSP page as shown in Figure 19.

Bonus Calculation

Enter social security Number:

Enter Multiplier:

Social security number retrieved: 777777777

Bonus Amount retrieved: 0.0

Error messages: Duplicate primary key

Figure 19 Duplicate Key Error

Create bonus.jsp

The code for `bonus.jsp` is fairly straight forward because the code to look up the session bean and calculate the bonus is now in the JavaBean. The first part of the file contains the HTML code to create the form. The code to pass the HTML form data to the JavaBean is in the second part of the file. The complete `bonus.jsp` file appears below. Look it over before going on to the discussion of its scriptlet and JSP-specific tags for interacting with the JavaBean.

```
<HTML>
<BODY BGCOLOR = "WHITE">
<HEAD>
<TITLE>Bonus Calculation</TITLE>
</HEAD>

<BLOCKQUOTE>
<H3>Bonus Calculation</H3>
```

```
<!--ACTION parameter calls this page-->
<FORM METHOD="GET" ACTION="bonus.jsp">

<P>
Enter social security Number:
<P>
<INPUT TYPE="TEXT" NAME="SOCSEC"></INPUT>
<P>

Enter Multiplier:
<P>
<INPUT TYPE="TEXT" NAME="MULTIPLIER"></INPUT>

<P>
<INPUT TYPE="SUBMIT" VALUE="Submit">
<INPUT TYPE="RESET">
</FORM>

<!--Scriptlet and JavaBeans Tags start here -->
<jsp:useBean id = "jbonus" class = "JBonusBean"/>

<%! String sMult, ssec; %>
<%
    sMult = request.getParameter("MULTIPLIER");
    ssec = request.getParameter("SOCSEC");
%>

<jsp:setProperty name = "jbonus" property="strMult" value="<%=sMult%"/>
<jsp:setProperty name = "jbonus" property="socsec" value="<%=ssec%"/>

Social security number retrieved:
<jsp:getProperty name="jbonus" property="socsec"/>

<P>
Bonus Amount retrieved:
<jsp:getProperty name="jbonus" property="bonusAmt"/>

<P>
Error messages:
<jsp:getProperty name = "jbonus" property="message"/>

</BLOCKQUOTE>

</BODY>
</HTML>
```

Specify the JavaBean

The following HTML tag specifies the JavaBean being used in this example. The `id` parameter defines an alias to use to reference the JavaBean, and the `class` parameter specifies the JavaBeans class. In this example the `id` is `jbonus` and the `class` is `JBonusBean`.

```
<jsp:useBean id = "jbonus" class = "JBonusBean"/>
```

Get the Data

The following JSP scriptlets retrieve the user-supplied data from the HTML form input fields. The multiplier is stored in the `sMult` String variable, and the social security number is stored in the `ssec` String variable.

```
<%! String sMult, ssec; %>
<%
sMult = request.getParameter("MULTIPLIER");
ssec = request.getParameter("SOCSEC");
%>
```

Pass the Data to the JavaBean

The following HTML tags set two properties in the JavaBean. A property is a private field in the JavaBean class. The first line uses the `jsp:setProperty` name tag to set the `strMult` field in the `JBonusBean` class (aliased by the `jbonus` `id`) to the value stored in the `sMult` variable. The second line performs a similar operation for the `socsec` field in the `JBonusBean` class.

```
<jsp:setProperty name = "jbonus" property="strMult" value="<%=sMult%"/>
<jsp:setProperty name = "jbonus" property="socsec" value="<%=ssec%"/>
```

The `value="<%=ssec%>"` expression sends the data contained in the `ssec` variable to the `socsec` field in the JavaBean.

Retrieve Data from the JavaBean

Retrieving data from a JavaBean is similar to sending data to it. You use the `jsp:getProperty` name tag and indicate the property (private field) whose data you want to get. The following `getProperty` name tag retrieves the data stored in the `socsec` private field of the `JBonusBean` class (aliased by the `jbonus` `id`).

```
Social security number retrieved:
<jsp:getProperty name="jbonus" property="socsec"/>
```

The following tags perform similar operations for the `bonusAmt` and `message` fields in the `JBonusBean` class.

```

<P>
Bonus Amount retrieved:
<jsp:getProperty name="jbonus" property="bonusAmt" />

<P>
Error messages:
<jsp:getProperty name = "jbonus" property="message" />

```

Create the JavaBeans Class

A JavaBeans™ class (or bean for short) looks just like any ordinary Java™ programming language class. But to be a bean, a JavaBeans class must follow a set of simple naming and design conventions as outlined in the JavaBeans specification. Because beans follow the JavaBeans specification, they can be accessed and managed by other programs and tools that follow the same conventions.

In the Create `bonus.jsp` (page 76) section, HTML tags and JSP scriptlets are used to get and set data in the private fields of the `JBonusBean` class. This is possible because the `JBonusBean` class follows the JavaBeans naming and design conventions.

This section describes the `JBonusBean` code and gives you a very simple introduction to JavaBeans technology as it is used with JSP pages. Visit the JavaBeans home page at <http://java.sun.com/beans/index.html> for further information on JavaBeans technology.

Here is the `JBonusBean` class in its entirety. A discussion of its pertinent parts follows.

```

import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import Beans.*;

public class JBonusBean {
    private String strMult, socsec, message;
    private double bonusAmt;
    CalcHome homecalc;

    public JBonusBean() {
        try{
            InitialContext ctx = new InitialContext();
            Object objref = ctx.lookup("calcs");
            homecalc = (CalcHome)
                PortableRemoteObject.narrow(
                    objref, CalcHome.class);
        } catch (javax.naming.NamingException e) {
            e.printStackTrace();
        }
    }
}

```

```
public double getBonusAmt() {
    if(strMult != null){
        Integer integerMult = new Integer(strMult);
        int multiplier = integerMult.intValue();
        try {
            double bonus = 100.00;
            Calc theCalculation = homecalc.create();
            Bonus theBonus = theCalculation.calcBonus(
                multiplier, bonus, socsec);
            Bonus record = theCalculation.getRecord(
                socsec);
            bonusAmt = record.getBonus();
            socsec = record.getSocSec();
        } catch (javax.ejb.DuplicateKeyException e) {
            message = e.getMessage();
        } catch (javax.ejb.CreateException e) {
            e.printStackTrace();
        } catch (java.rmi.RemoteException e) {
            e.printStackTrace();
        }
        return this.bonusAmt;
    } else {
        this.bonusAmt = 0;
        this.message = "None.";
        return this.bonusAmt;
    }
}

public String getMessage(){
    return this.message;
}
public String getSocsec(){
    return this.socsec;
}
public String getStrMult(){
    return this.strMult;
}
public void setSocsec(String socsec) {
    this.socsec = socsec;
}
public void setStrMult(String strMult) {
    this.strMult = strMult;
}
}
```


Bean Properties

Properties define the data that a JavaBean makes accessible to other programs and tools through get and set methods. The data might do things such as define the JavaBeans appearance or behavior, or be used in or the result of a series of calculations and computations. Properties are actually private class fields that should always be private and only accessible through get and set methods.

The following code segment shows the private properties for the `JBonusBean` class. The `JBonusBean` class has a corresponding `get<property>` method for each field and corresponding `set<property>` methods for the `strMult` and `socsec` fields.

```
public class JBonusBean {
    private String strMult, socsec, message;
    private double bonusAmt;
```

Constructor

The `JBonusBean` constructor looks up the session Bean.

```
public JBonusBean() {
    try{
        InitialContext ctx = new InitialContext();
        Object objref = ctx.lookup("calcs");
        homecalc = (CalcHome)
            PortableRemoteObject.narrow(
                objref, CalcHome.class);
    } catch (javax.naming.NamingException e) {
        e.printStackTrace();
    }
}
```

Set Methods

`JBonusBean` has two setter methods (methods prefixed with the word `set`). Setter methods set properties (private fields) to specified values. The two setter methods are `setSocsec` and `setStrMult` for setting the `socsec` and `strMult` private fields (JavaBean properties).

In this example, the values used to set the `socsec` and `strMult` properties come from the `setProperty` name tags in the JSP page. The J2EE server uses the information supplied in the following `setProperty` name tags to locate the corresponding set methods in the `JBonusBean` (aliased by the `jbonus` id):

```
<jsp:setProperty name = "jbonus" property="strMult" value="<%=sMult%"/>
<jsp:setProperty name = "jbonus" property="socsec" value="<%=ssec%"/>
```

In the `JBonusBean` class, the `set<property>` methods follow naming conventions so the J2EE server can map the `setProperty` name tags in the JSP file to the correct `set<property>` methods to pass the data from the JSP page to the JavaBean.

With setter methods, the method name consists of the word `set` and the property name. The property name is the name of one of the `JBonusBean` private fields. While field names begin with a lowercase letter by convention, the second word in a method name is always capitalized. So to set the `socsec` private field, the method name is `setSocsec`. The J2EE server maps the uppercase `Socsec` in the method name to the lowercase `socsec` field. Setter methods have no return value and have one argument of the appropriate type.

```
public void setSocsec(String socsec) {
    this.socsec = socsec;
}
public void setStrMult(String strMult) {
    this.strMult = strMult;
}
```

Get Methods

`JBonusBean` has four getter methods (methods prefixed with the word `get`). Getter methods `get` and return property values (private field values). The four getter methods are `getBonusAmt`, `getMessage`, `getSocsec`, and `getStrMult` for returning data from the `bonusAmt`, `message`, `socsec`, and `strMult` private fields (JavaBean properties).

In this example, the values used to set the `bonusAmt` and `message` fields come from the `getBonusAmt` method. The JSP page retrieves data from the `JBonusBean` properties using the following `getProperty` name tags. The JSP page retrieves only the values it is interested in, so you might notice that although there is a `JBonusBean` property for the multiplier (the `strMult` field), that value is not retrieved by the JSP page.

```
Social security number retrieved:
<jsp:getProperty name="jbonus" property="socsec"/>
<P>
Bonus Amount retrieved:
<jsp:getProperty name="jbonus" property="bonusAmt"/>
<P>
Error messages:
<jsp:getProperty name = "jbonus" property="message"/>
```

Getter methods follow the same naming conventions as setter methods so the JSP page can retrieve data from the `JBonusBean`. Getter methods always have a return value and no arguments. You might notice that although the `getBonusAmt` method sets property values and does not really need to return a value in this example, it returns `this.bonusAmt` to avoid a runtime J2EE server error.

The `getBonusAmt` method uses an `if-else` statement to handle the case where no `strMult` value is supplied. When the JSP page is first loaded, the end user has not supplied any data, but all tags and scriptlets on the page are executed anyway. In this event, the data value for the `strMult` property passed to `JBonusBean` is `null`, which results in a `null` multiplier and a `null` `bonusAmt` value. A runtime server error occurs when the JSP page gets and tries to display the `null` `bonusAmt` value. To prevent this runtime error, `bonusAmt` is set to 0 in the event a `null` `strMult` value is received from the JSP page.

```
public double getBonusAmt() {
    if(strMult != null){
        Integer integerMult = new Integer(strMult);
        int multiplier = integerMult.intValue();
        try {
            double bonus = 100.00;
            Calc theCalculation = homecalc.create();
            Bonus theBonus = theCalculation.calcBonus(
                multiplier, bonus, socsec);
            Bonus record = theCalculation.getRecord(
                socsec);
            bonusAmt = record.getBonus();
            socsec = record.getSocSec();
        } catch (javax.ejb.DuplicateKeyException e) {
            message = e.getMessage();
        } catch (javax.ejb.CreateException e) {
            e.printStackTrace();
        } catch (java.rmi.RemoteException e) {
            e.printStackTrace();
        }
        return this.bonusAmt;
    } else {
        this.bonusAmt = 0;
        this.message = "None.";
        return this.bonusAmt;
    }
}
public String getMessage(){
```

```
        return this.message;
    }
    public String getSocsec(){
        return this.socsec;
    }
    public String getStrMult(){
        return this.strMult;
    }
    public void setSocsec(String socsec) {
        this.socsec = socsec;
    }
    public void setStrMult(String strMult) {
        this.strMult = strMult;
    }
}
```

Start the Platform and Tools

To run this example, you need to start the J2EE server, the Deploy tool, and Cloudscape database. In different windows, type the following commands:

```
j2ee -verbose
deploytool
cloudscape -start
```

If that does not work, type this from the J2EE directory:

Unix

```
j2sdkee1.2.1/bin/j2ee -verbose
j2sdkee1.2.1/bin/deploytool
j2sdkee1.2.1/bin/cloudscape -start
```

Windows

```
j2sdkee1.2.1\bin\j2ee -verbose
j2sdkee1.2.1\bin\deploytool
j2sdkee1.2.1\bin\cloudscape -start
```

Remove the WAR File

Because you are adding a completely new class to the application, you have to delete the War file from the previous lesson and create a new one.

Local Applications:

- Click the `2BeansApp` icon so you can see its application components.
- Select `BonusWar` so it is outlined and highlighted.
- Select `Delete` from the **Edit** menu.

Create New WAR File

File menu:

- Select `New Web Component`.

Introduction:

- Read and Click `Next`.

War File General Properties:

- Specify `BonusWar` for the display name.
- Click `Add`.
- In the next window, go to the `ClientCode` directory, and add `bonus.jsp`.
- Click `Next`, go to the `ClientCode` directory, add `JBonusBean.class`
- Click `Finish`.

Note: Make sure you add `bonus.jsp` before you add `JBonusBean.class`.

War File General Properties:

- Click `Next`.

Choose Component Type:

- Make `Bonus.jsp` the JSP filename
- Make sure `Describe a JSP` is selected.
- Click `Next`.

Component General Properties:

- Make the display name `BonusJSP`.
- Click `Finish`.

Inspecting window:

- Select `Web Context`

- Specify `JSPRoot`.

Verify and Deploy the J2EE Application

Before you deploy the application, it is a good idea to run the verifier. The verifier will pick up errors in the application components such as missing enterprise bean methods that the compiler does not catch.

Verify:

- With `2BeansApp` selected, choose `Verifier` from the `Tools` menu.
- In the dialog that pops up, click `OK`. The window should tell you there were no failed tests. That is, if you used the session bean code provided for this lesson.
- Close the verifier window because you are now ready to deploy the application.

Note: In the Version 1.2.1 software you might get a `tests app.WebURI` error. This means the deploy tool did not put a `.war` extension on the `WAR` file during `WAR` file creation. This is a minor bug and the J2EE application deploys just fine in spite of it.

Deploy:

- From the `Tools` menu, choose `Deploy Application`. A **Deploy BonusApp** dialog box pops up.
- Verify that the `Target Server` selection is either `localhost` or the name of the host running the J2EE server.

Note: Do not check the `Return Client Jar` box. The only time you need to check this box is when you deploy a stand-alone application for the client program. This example uses an `HTML` and `JSP` page so this box should not be checked. Checking this box creates a `JAR` file with deployment information needed by a stand-alone application.

- Click `Next`. Make sure the `JNDI` names show `calcs` for `CalcBean` and `bonus` for `BonusBean`. If they do not show these names, type them in yourself, and press the `Return` key.
- Click `Next`. Make sure the `Context Root` name shows `JSPRoot`. If it does not, type it in yourself and press the `Return` key.
- Click `Next`.
- Click `Finish` to start the deployment. A dialog box pops up that displays the status of the deployment operation.
- When it is complete, the three bars on the left will be completely shaded as shown in Figure 20. When that happens, click `OK`.

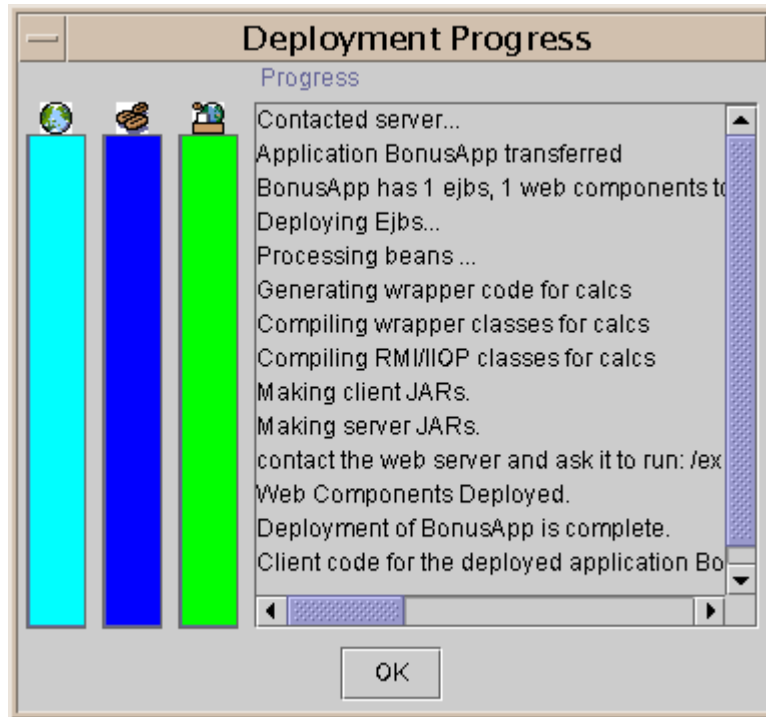


Figure 20 Deploy Application

Run the J2EE Application

The web server runs on port 8000 by default. To open the `bonus.jsp` page point your browser to `http://localhost:8000/JSPRoot/bonus.jsp`, which is where the Deploy tool put the JSP page.

- Fill in a social security number and multiplier
- Click the `Submit` button. `Bonus.jsp` processes your data and returns an HTML page with the bonus calculation on it.

See About the Example (page 74) for screen captures showing the application in action.

More Information

Visit the JavaBeans home page at `http://java.sun.com/beans/index.html` for further information on JavaBeans technology.

Lesson 6

Extensible Markup Language (XML)

eXtensible Markup Language (XML) is a language for representing and describing text-based data so the data can be read and handled by any program or tool that uses XML APIs. Programs and tools can generate XML files that other programs and tools can read and handle.

For example, a company might use XML to produce reports so different parties who receive the reports can handle the data in a way that best suits their needs. One party might put the XML data through a program to translate the XML to HTML so it can post the reports to the web, another party might put the XML data through a tool to produce a stockholder booklet, and yet another party might put the XML data through a tool to create a marketing presentation. Same data, different needs, and an array of platform-independent programs and tools to use the same data in any number of different ways. These highly flexible and cost-effective capabilities are available through XML tags, Document Type Definitions (DTDs) also known as XML schemas, and XML APIs.

This lesson adapts the example from Lesson 5 Adding JavaBeans Technology to the Mix (page 73) so the JavaBean class uses XML APIs to print a simple report where the data is marked with XML tags.

- Marking and Handling Text (page 90)
- Change the JavaBean Class (page 90)
- The APIs (page 95)
- Update and Run the Application (page 96)
- More Information (page 96)

Marking and Handling Text

With XML you define markup tags to represent the different elements of data in a text file. For example, if you have a text file that consists of a short article, you define XML tags to represent the title, author, first level heads, second level heads, bullet lists, article text, and so on. Once the data is represented by XML tags, you can create a Document Type Definition (DTD) and/or eXtensible Style sheet Language (XSL) file to describe how you want the data handled.

- XSL styles let you do things like map XML to HTML. For example, you can define an XML title tag to represent the title of an article, and create an XSL file that maps the XML title tag to the HTML H1 heading tag for display to the end user.
- A DTD (also known as an XML schema) contains specifications that allow other programs to validate the structure of an XML file to ensure the tagged data is in the correct format. For example, a DTD for an article might allow one title tag, but zero or more first and second level heads.

Any program capable of parsing XML can check for well-formed XML tags, and any program capable of applying XSL styles or DTD specifications to XML data can handle the tagged data intelligently. For example, if an article has two title tags, but the DTD allows only one, the program returns an error. Checking an XML document against a DTD is what is known as verification.

The nice thing about XML is the tagging is separate from the style sheet and DTD. This means you can have one XML document and one to many style sheets or DTDs. Different style sheets let you have a different presentation depending on how the document is used. For example, an article on XML can have a style sheet for the different web sites where it is to be published so it will blend with the look and feel of each site.

The current J2EE release does not have an eXtensible Style sheet Language Transformation (XSLT) engine so it is not currently possible to use a style sheet to do things such as transform an XML document into HTML for display.

Change the JavaBean Class

In this lesson, a `genXML` method is added to the `JBonusBean` class to generate the XML document shown below. A description of the code to generate this file comes after the discussion here of the XML document tags and structure.

```
<?xml version="1.0"?>
<report>
  <bonusCalc ssnnum="777777777" bonusAmt="300.0" />
</report>
```

XML Prolog

The `<?xml version="1.0" ?>` line is the XML prolog. An XML file should always start with a prolog that identifies the document as an XML file. The prolog is not required and is read only by humans, but it is good form to include it. Besides version information, the prolog can also contain encoding and standalone information.

- **Encoding information:** indicates the character set used to encode the document data. Uncompressed Unicode is shown as `<?xml version="1.0" encoding="UTF-8" ?>`. The Western European and English language character set is indicated by:
`<?xml version="1.0" encoding="ISO-8859-1" ?>`.
- **Standalone information:** indicates if this document uses information in other files. For example, an XML document might rely on a style sheet for information on how to create the user interface in HTML, or a DTD for valid tag specifications.

Document Root

The `<report>` tag is the first XML tag in this file. It is the top-level XML tag and marks the beginning of the document data. Another name for this level tag is `root`. XML tags have a matching end tag, so the end of this document has the corresponding `</report>` tag to close the pair.

You can give XML tags any name you want. This example uses `report` because the XML file is a bonus report. It could just as well be named `<root>` or `<begin>` or whatever. The name takes on meaning in the style sheet and DTD because that is where you assign specifications to tags by their names.

Child Nodes

The `<bonusCalc>` tag represents the bonus report. This tag is a child node that is added to the root. It uses attributes to specify the social security number and bonus amount values (`ssnum` and `bonusAmt`). You can define a DTD to check that the `bonusCalc` tag has the `ssnum` attribute and `bonusAmt` attributes, and have your program raise an error if an attribute is missing or if attributes are present that should not be there.

```
<bonusCalc ssnum="777777777" bonusAmt="300.0" />
```

Other XML Tags

There are a number of ways to tag data. This example uses empty tags, which are tags that do not enclose data, use attributes to specify data, and are closed with a slash. The following empty tag from this example, could be created so the data is enclosed by XML tags instead. The XML parser checks that all data enclosed by data has what are called well-formed tags. Well-formed tags consist of an opening tag and a closing tag as shown in the well-formed tag example below.

Empty tag:

```
<bonusCalc ssnum="777777777" bonusAmt="300.0" />
```

Well-formed tags:

```
<bonusCalc>
  <ssnum>"777777777"</ssnum>
  <bonusAmt>300.0</bonusAmt>
</bonusCalc>
```

XML comment tags look just like HTML comment tags.

```
<!-- Bonus Report -->
<bonusCalc ssnum="777777777" bonusAmt="300.0" />
```

Processing Instructions give commands or information to an application that is processing the XML data. Processing instructions have the format `<? target instructions?>` where `target` is the name of the application doing the processing, and `instructions` embodies the information or commands for the application to process. The prolog is an example of a processing instruction, where `xml` is the target and `version="1.0"` embodies the instructions. Note that the target name `xml` is reserved for XML standards.

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

You can also use processing instructions to do things like distinguish between different versions of a presentation such as the high-level executive version and the technical version.

JavaBean Code

The `JBonusBean` class for this lesson has `import` statements for creating the XML document, handling errors, and writing the document out to the terminal. This lesson writes the XML output to the terminal to keep things simple. The XML output could just as well be written to a file, but you would need to configure your browser to use Java Plug-In and include a security policy file granting the JavaBean code permission to write to the file.

To generate the XML file for this lesson, you need to import the `ElementNode` and `XmlDocument` classes. You also need the `StringWriter` and `IOException` classes to write the XML data to the terminal.

```
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import Beans.*;
import java.io.StringWriter;
import java.io.IOException;
import com.sun.xml.tree.ElementNode;
import com.sun.xml.tree.XmlDocument;
```

This version of the `JBonusBean` class has one more instance variables. The session bean's remote interface, `theCalculation`, needs to be accessed from the `getBonusAmt` and `genXML` methods. This is because `genXML` reads the database to generate XML for all records stored in the database and has to be able to access the session bean's `getRecord` method.

```
Calc theCalculation;
```

The `JBonusBean.genXML` method is called from the `getBonusAmt` method after the processing completes in the event `strMult` is not `null`. The first thing this method does is create an `XMLDocument` object and the root node, and adds the root to the document. The root node represents the top-level point in the document hierarchy (or tree) and is the point at which processing begins.

```
private void genXML(){
    Bonus records = null;
    //Create XML document
    XmlDocument doc = new XmlDocument();
    //Create node
    ElementNode root = (ElementNode)
        doc.createElement("report");
    //Add node to XML document
    doc.appendChild(root);
}
```

The try and catch block that comes next, gets the record out of the database, retrieves the bonus amount and social security number from the record, converts the bonus amount to a string, creates a child node (`bonusCalc`), and adds the social security number and bonus amount to the `bonusCalc` child node as attributes. The child node represents the second level in the document hierarchy or tree, and the attributes represent the third level.

```
try{
    //Get database record
    records = theCalculation.getRecord(socsec);
    //Retrieve the social security number from record
    String ssRetrieved = records.getSocSec();
    //Retrieve bonus amount from record
    double bRetrieved = records.getBonus();
    //Convert double to string
    Double bonusObj = new Double(bRetrieved);
    String bString = bonusObj.toString();
    //Create child node
    ElementNode bonusCalc = (ElementNode)
        doc.createElement("bonusCalc");
    //Add attributes to child node
    bonusCalc.setAttribute("ssnum", ssRetrieved);
    bonusCalc.setAttribute("bonusAmt", bString);
    //Add child node to root
    root.appendChild(bonusCalc);
} catch (java.rmi.RemoteException e) {
    e.printStackTrace();
}
```

The last part of the `genXML` method creates a `StringWriter` object, writes the document hierarchy or tree to the `StringWriter` object, and writes the `StringWriter` object to the terminal.

```
try{
    StringWriter out = new StringWriter();
    doc.write(out);
    System.out.println(out);
} catch (java.io.FileNotFoundException fe) {
    System.out.println("Cannot write XML");
} catch (IOException ioe) {
    System.out.println("cannot write XML");
}
```

The hierarchy or tree structure for the XML document is called the Document Object Model (DOM). Figure 21 shows a simplified representation of the DOM for this lesson's example. The API calls in the `genXML` method create the DOM and you can make API calls to access the DOM to do such things as add, delete, and edit child nodes, or validate the DOM against a DTD. You can also create a DOM from an XML file.

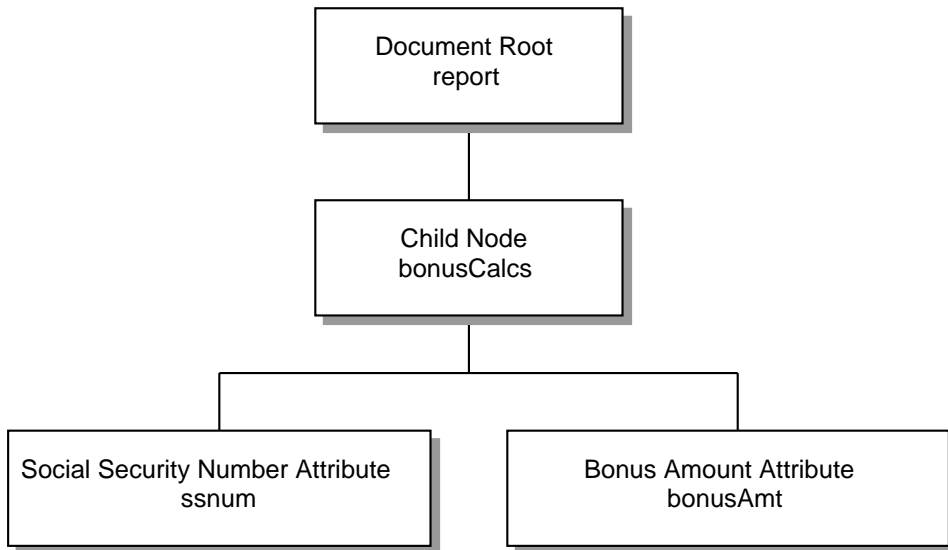


Figure 21 Document Object Model (DOM)

The APIs

The `j2ee.jar` file that comes with your J2EE installation provides APIs for parsing and manipulating XML data. The JAR file currently provides SAX, DOM, and J2EE XML APIs. You can use whichever API best suits your needs because as shown in Figure 22, XML text is independent of the platform and language of its creation..

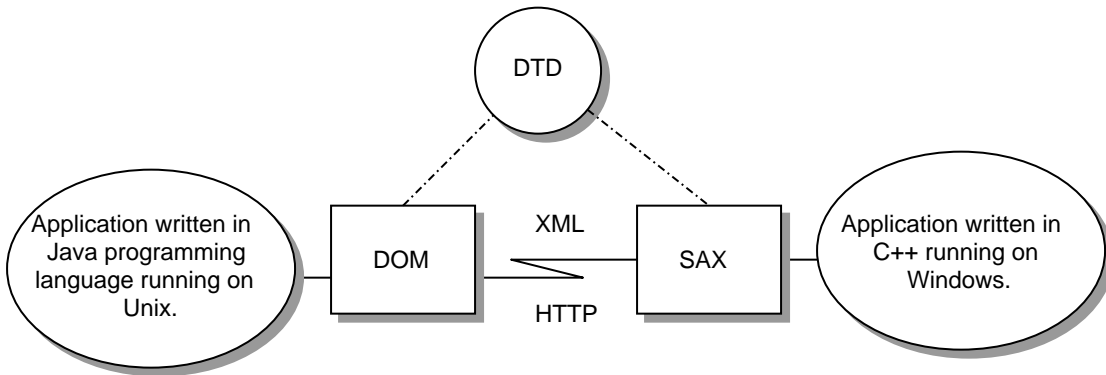


Figure 22 Platform and Language Neutral Text

SAX and DOM

The SAX API is an event-driven, serial-access mechanism that does element by element processing.

The DOM API provides a relatively familiar tree structure of objects. You can use the DOM API to manipulate the hierarchy of application objects it encapsulates. The DOM API is ideal for interactive applications because the entire object model is present in memory, where it can be accessed and manipulated by the user.

Constructing the DOM requires reading the entire XML structure and holding the object tree in memory, so it is much more CPU and memory intensive. For that reason, the SAX API will tend to be preferred for server-side applications and data filters that do not require an in-memory representation of the data.

Note: You can find more information on the DOM and SAX APIs at this location:

http://java.sun.com/xml/docs/tutorial/overview/3_apis.html

J2EE

The platform-independent J2EE XML APIs use a DOM tree and provide a wealth of methods for manipulating the DOM hierarchy. The J2EE XML APIs are in the package `com.sun` and were used in this lesson's example. Please note that these APIs are subject to change.

Update and Run the Application

Because all you have done in this lesson is change the `JBonusBean` class, you can simply update and redeploy the application.

- **Local Applicatons Window:** Highlight the `2BeansApp` application.
- **Tools Menu:** Select Update and Redeploy Application.

Note: The `BonusApp` application from the previous lesson is automatically uninstalled

The web server runs on port 8000 by default. To open the `bonus.jsp` page point your browser to `http://localhost:8000/JSPRoot/bonus.jsp`, which is where the Deploy tool put the JSP page.

- Fill in a social security number and multiplier
- Click the `Submit` button. `Bonus.jsp` processes your data and returns an HTML page with the bonus calculation on it.

More Information

There is a lot of information about XML on the Web that you can access with a good search engine. A very good web site is `www.xml.com`. The `java.sun.com` site has an XML tutorial at `http://java.sun.com/xml/docs/tutorial/index.html`.

Lesson 7

JDBC Technology and Bean-Managed Persistence

Up to this point, the example J2EE application has written data to and read data from the underlying Cloudscape database without your writing and SQL code. This is because the container has been handling data storage and retrieval on behalf of the entity bean. Container-managed persistence is the term used to describe the situation where the container handles data storage and retrieval. This lesson shows you how to override the default container-managed persistence and implement bean-managed persistence.

Bean-managed persistence is when you override container-managed persistence and implement entity or session bean methods to use the SQL commands you provide. Bean-managed persistence can be useful if you need to improve performance or map data in multiple beans to one row in a database table.

This lesson changes the entity bean in the example J2EE application to use bean-managed persistence.

- [Bean Lifecycle \(page 98\)](#)
- [Change the BonusBean Code \(page 99\)](#)
- [Change the CalcBean and JBonusBean Code \(page 106\)](#)
- [Create the Database Table \(page 107\)](#)
- [Remove the JAR File \(page 109\)](#)
- [Verify and Deploy the Application \(page 111\)](#)
- [Run the Application \(page 112\)](#)
- [More Information \(page 113\)](#)

Bean Lifecycle

The BonusBean (page 30) section in Lesson 3 shows the container-managed BonusBean class. The only methods with implementations are `getBonus` to return the bonus value, `getSocSec` to return the social security number, and `ejbCreate` to create an entity bean with the `bonus` and `socsec` values passed to it. The container takes care of such things as creating a row in the database table for the data, and ensuring the data in memory is consistent with the data in the table row. With bean-managed persistence, you have to implement all of this behavior yourself, which means adding JDBC™ and SQL code, and implementing the empty methods in the container-managed example.

A session or an entity bean consists of business methods and lifecycle methods. In the example, `CalcBean` has two business methods, `calcBean` and `getRecord`, and `BonusBean` has two business methods, `getBonus` and `getSocsec`. Both `CalcBean` and `BonusBean` have the following lifecycle methods. Business methods are called by clients and lifecycle methods are called by the bean's container.

- `setEntityContext`: The container calls this method first to pass an entity context object to the entity bean. The entity context is dynamically updated by the container so even if the entity bean is invoked by many clients over its lifetime, the context contains current data for each invocation. A session bean has a corresponding `setSessionContext` method that performs a similar function as the `setEntityContext` method.
- `ejbCreate`: The container calls this method when a client calls a create method in the bean's home interface. For each create method in the home interface, the bean has a corresponding `ejbCreate` method with the same signature (parameters and return value).
- `ejbPostCreate`: The container calls this method after the `ejbCreate` method completes. There is an `ejbPostCreate` method for every `ejbCreate` method that takes the same arguments as its corresponding create method. However, `ejbPostCreate` has no return value. Use `ejbPostCreate` to implement any special processing needed after the bean is created, but before it becomes available to the client. Leave this method empty if no special processing is needed.
- `ejbRemove`: The container calls this method when a client calls a remove method in the bean's home interface. The example J2EE application for this tutorial does not include a remove method in the home interface.
- `unsetEntityContext`: The container calls this method after the `ejbRemove` has been called to remove the entity bean from existence. Only entity beans have an `unsetEntityContext` method. A session bean does not have a corresponding `unsetSessionContext` method.
- `ejbFindByPrimaryKey`: The container calls this method when a client calls the `findByPrimaryKey` method in the bean's home interface. For each find method in the home interface, the bean has a corresponding `ejbFind<type>` method with the same signature (parameters and return value).

- `ejbLoad` and `ejbStore`: The container calls these methods to synchronize the bean's state with the underlying database. When a client sets or gets data in the bean such as in the case of a get method, the container calls `ejbStore` to send the object data to the database and calls `ejbLoad` to read it back in again. When a client calls a finder method, the container calls `ejbLoad` to initialize the bean with data from the underlying database.
- `ejbActivate` and `ejbPassivate`: The container calls these methods to activate and passivate the bean's state. Activation and passivation refer to swapping a bean in and out of temporary storage to free memory, which might occur if a given bean has not been called by a client in a long time. Implementations for `ejbPassivate` might include things like closing connections or files used by the bean, and for `ejbActivate` might include things like reopening those same connections or files.

Change the BonusBean Code

This section walks through the bean-managed persistence `BonusBean` code. The first thing you will notice is that there is a lot more code here than for the container-managed persistence version.

Import Statements

The `InitialContext`, `DataSource`, and `Connection` interfaces are imported for establishing a connection to the database. The `PreparedStatement` interface is imported to be used as a template to create a SQL request. The `ResultSet` interface is imported to manage access to data rows returned by a query. The `FinderException` and `SQLException` classes are imported to handle lookup and database access exceptions.

```
package Beans;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import javax.ejb.FinderException;
import java.sql.SQLException;
```

Instance Variables

The instance variables added to this lesson let you establish and close database connections. The string `java:comp/env/jdbc/BonusDB` indicates the resource reference name, which you also specify when you add the entity bean to the J2EE application using the Deploy tool. In this example, the resource reference is an alias to the Cloudscape database (`CloudscapeDB`) where the table data is stored.

Later, you will create the `BONUS` table in the `CloudscapeDB`, and during deployment, you will map `jdbc/BonusDB` to `jdbc/CloudscapeDB`.

```
public class BonusBean implements EntityBean {
    private EntityContext context;
    private Connection con;
    private String dbName =
        "java:comp/env/jdbc/BonusDB";
    private InitialContext ic = null;
    private PreparedStatement ps = null;
    private double bonus;
    private String socsec;
```

Business Methods

The business methods have not changed for this lesson except for calls to `System.out.println`, which let you see the order in which business and lifecycle methods are called at runtime.

```
public double getBonus() {
    System.out.println("getBonus");
    return this.bonus;
}
public String getSocSec() {
    System.out.println("getSocSec");
    return this.socsec;
}
```

LifeCycle Methods

These methods include calls to `System.out.println` so you can see the order in which business and lifecycle methods are called at runtime.

ejbCreate

The `ejbCreate` method signature for this lesson throws `RemoteException` and `SQLException` in addition to `CreateException`. `SQLException` is needed because the `ejbCreate` method for this lesson provides its own SQL code (it does not rely on the container to provide it), and `RemoteException` is needed because this method performs remote access.

One thing to notice about this class is that it returns a `String` value which is the primary key, but the declaration for this method in the `home` interface expects to receive a `Bonus` class

instance. The container uses the primary key returned by this method to create the Bonus instance.

```

public String ejbCreate(double bonus, String socsec)
    throws RemoteException,
        CreateException,
        SQLException {
    this.socsec=socsec;
    this.bonus=bonus;

    System.out.println("Create Method");
    try {
//Establish database connection
        ic = new InitialContext();
        DataSource ds = (DataSource) ic.lookup(dbName);
        con = ds.getConnection();
//Use PreparedStatement to form SQL INSERT statement
//to insert into BONUS table
        ps = con.prepareStatement(
            "INSERT INTO BONUS VALUES ( ? , ?)");
//Set 1st PreparedStatement value marked by ? , with
//socsec and the 2nd value marked by ?) with bonus
        ps.setString(1, socsec);
        ps.setDouble(2, bonus);
        ps.executeUpdate();
    } catch (javax.naming.NamingException ex) {
        ex.printStackTrace();
    } finally {
//Close database connection
        ps.close();
        con.close();
    }
//Return primary key
    return socsec;
}

```

ejbPostCreate

This method has the same signature as `ejbCreate`, but no implementation because this simple example performs no post create processing or initialization.

```

public void ejbPostCreate(double bonus,
    String socsec)
    throws RemoteException,
        CreateException,
        SQLException {
    System.out.println("Post Create");
}

```

ejbFindByPrimaryKey

The container-managed version of `BonusBean` did not include an `ejbFindByPrimaryKey` implementation because the container can locate database records by their primary keys if you specify container-managed persistence and provide the primary key field during deployment. In this lesson, `BonusBean` is deployed with bean-managed persistence so you must provide an implementation for this method and throw the `SQLException`. The container-managed version throws `RemoteException` and `FinderException` only.

If the find operation locates a record with the primary key passed to `ejbFindByPrimaryKey`, the primary key value is returned so the container can call the `ejbLoad` method to initialize `BonusBean` with the retrieved `bonus` and `socsec` data.

One thing to notice about this class is that it returns a `String` value which is the primary key, but the declaration for this method in the home interface expects to receive a `Bonus` class instance. The container uses the primary key returned by this method to create the `Bonus` instance.

```
public String ejbFindByPrimaryKey(String primaryKey)
    throws RemoteException, FinderException,
        SQLException {
    System.out.println("Find by primary key");
    try {
//Establish database connection
        ic = new InitialContext();
        DataSource ds = (DataSource) ic.lookup(dbName);
        con = ds.getConnection();
//Use PreparedStatement to form SQL SELECT statement
//to select from BONUS table
        ps = con.prepareStatement(
            "SELECT socsec FROM BONUS WHERE socsec = ? ");
        ps.setString(1, primaryKey);
//Use ResultSet to capture SELECT statement results
        ResultSet rs = ps.executeQuery();
//If ResultSet has a value, the find was successful,
//and so initialize and return key
        if(rs.next()) {
            key = primaryKey;
        } else {
            System.out.println("Find Error");
        }
    } catch (javax.naming.NamingException ex) {
        ex.printStackTrace();
    } finally {
//Close database connection
        ps.close();
        con.close();
    }
//Return primary key
    return key;
}
```

ejbLoad

This method is called after a successful call to `ejbFindByPrimaryKey` to load the retrieved data and synchronize the bean data with the database data.

```

public void ejbLoad() {
    System.out.println("Load method");
    try {
//Establish database connection
        ic = new InitialContext();
        DataSource ds = (DataSource) ic.lookup(dbName);
        con = ds.getConnection();
//Use PreparedStatement to form SQL SELECT statement
//to select from BONUS table
        ps = con.prepareStatement(
            "SELECT * FROM BONUS WHERE SOCSEC = ?");
        ps.setString(1, this.socsec);
//Use ResultSet to capture SELECT statement results
        ResultSet rs = ps.executeQuery();
//If ResultSet has a value, the find was successful
        if(rs.next()){
            this.bonus = rs.getDouble(2);
        } else {
            System.out.println("Load Error");
        }
    } catch (java.sql.SQLException ex) {
        ex.printStackTrace();
    } catch (javax.naming.NamingException ex) {
        ex.printStackTrace();
    } finally {
        try {
//Close database connection
            ps.close();
            con.close();
        } catch (java.sql.SQLException ex) {
            ex.printStackTrace();
        }
    }
}
}

```

ejbStore

This method is called when a client sets or gets data in the bean to send the object data to the database and keep the bean data synchronized with the database data.

```

public void ejbStore() {
    System.out.println("Store method");
    try {
//Establish database connection
        DataSource ds = (DataSource)ic.lookup(dbName);
        con = ds.getConnection();
//Use PreparedStatement to form SQL UPDATE statement

```

```

//to update BONUS table
    ps = con.prepareStatement(
        "UPDATE BONUS SET BONUS = ? WHERE SOCSEC = ?");
//Set 1st PreparedStatement value marked by ? with
//bonus and the 2nd value marked by ?) with socsec
    ps.setDouble(1, bonus);
    ps.setString(2, socsec);
    int rowCount = ps.executeUpdate();
} catch (javax.naming.NamingException ex) {
    ex.printStackTrace();
} catch (java.sql.SQLException ex) {
    ex.printStackTrace();
} finally {
    try {
//Close database connection
        ps.close();
        con.close();
    } catch (java.sql.SQLException ex) {
        ex.printStackTrace();
    }
}
}
}

```

ejbRemove

This method is called when a client calls a `remove` method on the bean's home interface. The JavaBean client in this example does not provide a `remove` method that a client can call to remove `BonusBean` from its container. Nevertheless, the implementation for an `ejbRemove` method is shown here. When the container calls `ejbRemove`, `ejbRemove` gets the primary key (`socsec`) from the `socsec` instance variable, removes the bean from its container, and deletes the corresponding database row.

```

public void ejbRemove()
    throws RemoteException {
    System.out.println("Remove method");
    try {
        DataSource ds = (DataSource)ic.lookup(dbName);
        con = ds.getConnection();
        ps = con.prepareStatement(
            "DELETE FROM BONUS WHERE SOCSEC = ?");
        ps.setString(1, socsec);
        ps.executeUpdate();
    } catch (java.sql.SQLException ex) {
        ex.printStackTrace();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    try {
        ps.close();
        con.close();
    } catch (java.sql.SQLException ex) {
        ex.printStackTrace();
    }
}
}

```


ejbActivate

When a bean has not been used in a long time, the container passivates it or moves it to temporary storage where the container can readily reactivate the bean in the event a client calls one of the bean's business methods. This method calls the `getPrimaryKey` method on the entity context so the primary key is available to clients querying the bean. When a query is made, the container uses the primary key to load the bean data.

```
public void ejbActivate() {
    System.out.println("Activate method");
    socsec = (String)context.getPrimaryKey();
}
```

ejbPassivate

When a bean has not been used in a long time, the container passivates it or moves it to temporary storage where the container can readily reactivate the bean in the event a client calls one of the bean's business methods. This method sets the `primary` key to `null` to free memory while the bean is in the passive state.

```
public void ejbPassivate() {
    System.out.println("Passivate method");
    socsec = null;
}
```

setEntityContext

This method is called by the container to initialize the bean's `context` instance variable. This is needed because the `ejbActivate` method calls the `getPrimaryKey` method on the `context` instance variable to move a passive bean to its active state.

```
public void setEntityContext(
    javax.ejb.EntityContext ctx){
    System.out.println("setEntityContext method");
    this.context = ctx;
}
```

unsetEntityContext

This method is called by the container to set the `context` instance variable to `null` after the `ejbRemove` method has been called to remove the entity bean from existence. Only entity beans have an `unsetEntityContext` method.

```
public void unsetEntityContext(){
    System.out.println("unsetEntityContext method");
    ctx = null;
}
}
```

Change the CalcBean and JBonusBean Code

Because `BonusBean` provides its own SQL code, the `CalcBean.calcbonus` method, which creates `BonusBean` instances, has to be changed to throw `java.sql.SQLException`. Here is one way to do make that change:

```
public class CalcBean implements SessionBean {
    BonusHome homebonus;

    public Bonus calcBonus(int multiplier,
                           double bonus, String socsec)
        throws RemoteException,
        SQLException,
        CreateException {

        Bonus theBonus = null;
        double calc = (multiplier*bonus);

        try {
            InitialContext ctx = new InitialContext();
            Object objref = ctx.lookup("bonus");
            homebonus = (BonusHome)
                PortableRemoteObject.narrow(
                    objref, BonusHome.class);
        } catch (Exception NamingException) {
            NamingException.printStackTrace();
        }

        //Store data in entity Bean
        theBonus=homebonus.create(calc, socsec);
        return theBonus;
    }
}
```

The `JBonusBean` class has to be changed to catch the `SQLException` thrown by `CalcBean.DuplicateKeyExcpetion` is a subclass of `CreateException`, so it will be caught by the `catch (javax.ejb.CreateException e)` statement.

```
public double getBonusAmt() {
    if(strMult != null){
        Integer integerMult = new Integer(strMult);
        int multiplier = integerMult.intValue();
        try {
            double bonus = 100.00;
            theCalculation = homecalc.create();
            Bonus theBonus = theCalculation.calcBonus(
                multiplier, bonus, socsec);
            Bonus record = theCalculation.getRecord(
                socsec);
            bonusAmt = record.getBonus();
            socsec = record.getSocSec();
        } catch (java.sql.SQLException e) {
            this.bonusAmt = 0.0;
        }
    }
}
```

```

        this.socsec = "000";
        this.message = e.getMessage();
    } catch (javax.ejb.CreateException e) {
        this.bonusAmt = 0.0;
        this.socsec = "000";
        this.message = e.getMessage();
    } catch (java.rmi.RemoteException e) {
        this.bonusAmt = 0.0;
        this.socsec = "000";
        this.message = e.getMessage();
    }
    genXML();
    return this.bonusAmt;
} else {
    this.bonusAmt = 0;
    this.message = "None.";
    return this.bonusAmt;
}
}

```

Create the Database Table

Because this example uses bean-managed persistence, you have to create the `BONUS` database table in the `CloudscapeDB` database. With container-managed persistence, the table is created for you.

To make things easy, the database table is created with two scripts: `createTable.sql` and `cloudTable.sh` (Unix) or `cloudTable.bat` (Windows/NT). For this example, the `createTable.sql` script goes in your `~/J2EE/Beans` directory, and the `cloudTable.sh` (Unix) or `cloudTable.bat` (Windows/NT) script goes in your `~/J2EE` directory.

To execute the scripts, go to the `Beans` directory and type the following:

Unix:

```
../cloudTable.sh
```

Windows/NT:

```
..\cloudTable.bat
```

createTable.sql

This file is provided in the code download for this lesson.

```

drop table bonus;

create table bonus
(socsec varchar(9) constraint pk_bonus primary key,
bonus decimal(10,2));

exit;

```

cloudTable.bat

This file is provided in the code download for this lesson.

```

rem cloudTable.bat
rem Creates BONUS table in CloudscapeDB.
rem
rem Place this script in ~\J2EE
rem To run: cd ~\J2EE\cloudTable.sh
rem
rem Change this next line to point to *your*
rem j2sdkeel.2.1 installation
rem
set J2EE_HOME=\home\monicap\J2EE\j2sdkeel.2.1
rem
rem Everything below goes on one line
java -Dij.connection.CloudscapeDB=
jdbc:rmi://localhost:1099/jdbc:cloudscape:
CloudscapeDB;create=true -Dcloudscape.system.home=
%J2EE_HOME%\cloudscape -classpath
%J2EE_HOME%\lib\cloudscape\client.jar;
%J2EE_HOME%\lib\cloudscape\tools.jar;
%J2EE_HOME%\lib\cloudscape\cloudscape.jar;
%J2EE_HOME%\lib\cloudscape\RmiJdbc.jar;
%J2EE_HOME%\lib\cloudscape\license.jar;
%CLASSPATH% -ms16m -mx32m
COM.cloudscape.tools.ij createTable.sql

```

cloudTable.sh

This file is provided in the code download for this lesson.

```

#!/bin/sh
#
# cloudTable.sh
# Creates BONUS table in CloudscapeDB.
#
# Place this script in ~\J2EE
# To run: cd ~\J2EE\cloudTable.sh
#
# Change this next line to point to *your*
# j2sdkeel.2.1 installation
#
J2EE_HOME=/home/monicap/J2EE/j2sdkeel.2

```

```
#
# Everything below goes on one line
java -Dij.connection.CloudscapeDB=jdbc:rmi:
//localhost:1099/jdbc:cloudscape:CloudscapeDB\;
create=true -Dcloudscape.system.home=
$J2EE_HOME/cloudscape -classpath
$J2EE_HOME/lib/cloudscape/client.jar:
$J2EE_HOME/lib/cloudscape/tools.jar:
$J2EE_HOME/lib/cloudscape/cloudscape.jar:
$J2EE_HOME/lib/cloudscape/RmiJdbc.jar:
$J2EE_HOME/lib/cloudscape/license.jar:
${CLASSPATH} -ms16m -mx32m
COM.cloudscape.tools.ij createTable.sql
```

Remove the JAR File

You have to update the bean JAR file with the new entity bean code. If you have both beans in one JAR file, you have to delete the **2BeansJar** and create a new one. The steps to adding `CalcBean` are the same as in *Create JAR with Session Bean* (page 54). The steps to adding `BonusBean` are slightly different and described here.

If you have the beans in separate JAR files, you have to delete the JAR file with `BonusBean` and create a new one as described here.

These instructions pick up at the point where you add the `BonusBean` interfaces and classes to the JAR file.

EJB JAR:

- Click Add (the one next to the **Contents** window).
- Toggle the directory so the Beans directory displays with its contents.
- Select `Bonus.class`
- Click Add.
- Select `BonusBean.class`
- Click Add.
- Select `BonusHome.class`
- Click Add.

Enterprise Bean JAR classes:

- Make sure you see `Beans/Bonus.class`, `Beans/BonusHome.class`, and `Beans/BonusBean.class` in the display.
- Click OK.

EJB JAR:

- Click Next.

General:

- Make sure `Beans.BonusBean` is the classname, `Beans.BonusHome` is the Home interface, and `Beans.Bonus` is the Remote interface.
- Enter `BonusBean` as the display name.
- Click **Entity**.
- Click `Next`.

Entity Settings:

- Select `Bean-managed persistence`.
- The primary key class is `java.lang.String`, Note that the primary key has to be a class type. Primitive types are not valid for primary keys.
- Click `Next`.

Environment Entries:

- Click `Next`. This simple entity bean does not use properties (environment entries).

Enterprise Bean References:

- Click `Next`.

Resource References:

- Click `Add`
- type `jdbc/BonusDB` in the first column under **Coded Name**. Make sure `Type` is `javax.sql.DataSource`, and **Authentication** is `Container`.
- Click `Next`.

Security:

- Click `Next`. This simple entity bean does not use security roles.

Transaction Management:

- Select `Container-managed transactions` (if it is not already selected).
- In the list below make `create`, `findByPrimaryKey`, `getBonus` and `getSocSec` required. This means the container starts a new transaction before running these methods. The transaction commits just before the methods end. You can find more information on these transaction settings in Chapter 6 of the Enterprise JavaBeans Developer's Guide.
- Click `Next`.

Review Settings:

- Click `Finish`.

Inspecting window:

- With `2BeansApp` selected, click **JNDI names**.
- Assign `calcs` to `CalcBean`, `bonus` to `BonusBean`, and `jdbc/Cloudscape` to `jdbc/BonusDB`.

Verify and Deploy the Application

Before you deploy the application, it is a good idea to run the verifier. The verifier will pick up errors in the application components such as missing enterprise bean methods that the compiler does not catch.

Note: If you get a Save error when you verify or deploy, shut everything down and restart the server and tools.

Verify:

- With `2BeansApp` selected, choose `Verifier` from the `Tools` menu.
- In the dialog that pops up, click `OK`. The window should tell you there were no failed tests.
- Close the verifier window because you are now ready to deploy the application.

Note: In the Version 1.2.1 software you might get a `tests app.WebURI` error. This means the deploy tool did not put a `.war` extension on the `WAR` file during `WAR` file creation. This is a minor bug and the `J2EE` application deploys just fine in spite of it.

Deploy:

- From the **Tools** menu, choose `Deploy Application`. A **Deploy BonusApp** dialog box pops up.
- Verify that the `Target Server` selection is either `localhost` or the name of the host running the `J2EE` server.
- Check the **Return Client Jar** box. Checking this box creates a `JAR` file with deployment information needed by the entity bean.
- Click `Next`.
- Make sure the `JNDI` names show for `calcs` for `CalcBean`, `bonus` for `BonusBean`, and `jdbc/Cloudscape` for `BonusDB`. If they do not, type the `JNDI` names in yourself, and press the `Return` key.
- Click `Next`. Make sure the `Context Root` name shows `JSPRoot`. If it does not, type it in yourself and press the `Return` key.
- Click `Next`.
- Click `Finish` to start the deployment. A dialog box pops up that displays the status of the deployment operation.
- When it is complete, click `OK`.

Run the Application

The web server runs on port 8000 by default. To open the `bonus.jsp` page point your browser to `http://localhost:8000/JSPRoot/bonus.jsp`, which is where the Deploy tool put the JSP page.

- Fill in a social security number and multiplier
- Click the `Submit` button. `Bonus.jsp` processes your data and returns an HTML page with the bonus calculation on it.

The J2EE server output might show the following message each time database access is attempted. The message means no user name and password were supplied to access the database. You can ignore this message because a user name and password are not required to access the Cloudscape database, and this example works just fine regardless of the message.

```
Cannot find principal mapping information for data source with JNDI name
jdbc/Cloudscape
```

Here is a cleaned up version of the J2EE server output (the above message was edited out).

```
setEntityContext method
Create Method
Post Create

setEntityContext method
Find by primary key
Load method

getBonus
Store method
Load method

getSocSec
Store method
Find by primary key
Load method

getSocSec
Store method
Load method

getBonus
Store method

<?xml version="1.0"?>
<report>
  <bonusCalc ssnum="777777777" bonusAmt="300.0" />
</report>
```


More Information

You can get more information on entity Beans and bean-managed persistence here:

<http://java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Entity.fm.html>

You can get more information on making database connections here:

<http://java.sun.com/j2ee/j2sdkee/techdocs/guides/ejb/html/Database.fm.html>

Index

A

- application assembly 16
- application components
 - editing information 22
 - working together 10
- application deployment 24, 58, 69, 86, 111
- application verification 23, 58, 68, 86
- avax.rmi.RemoteException 12

B

- bonus.html file 6
- BonusServlet code 6

C

- Cloudscape database 27
- container managed
 - persistence 30
 - transaction management 30
- Content pane 22
- context root
 - calling a servlet in an HTML form 6
 - specify 22
- create method 12, 28
- CreateException class 11

D

- deploy application 24, 58, 69, 86, 111
- deploy tool
 - assemble application 16
 - deploy application 24, 58, 69, 86, 111
 - described 15
 - editing information 22
 - verify application 23, 58, 68, 86
 - view application components 19
- deploytool command 14
- doGet method 7

E

- editing information 22
- ejbCreate method 12, 28
- EJBObject class 12
- entity Bean
 - container managed 30
 - defined 28

F

- findByPrimaryKey method 28

G

- getBonus method 29
- getSocSec method 29

H

- home interface
 - looking up 7
 - role of 11
- HTTP headers 7
- HttpServlet class 7

I

- IOException class 7

J

- J2EE application components
 - defined 4
- j2ee -verbose command 14
- java.io 7
- javax.naming 7
- javax.rmi 7
- javax.servlet 7
- javax.servlet.http 7
- JNDI name
 - how used 7
 - specify 22

L

looking up the home interface 7

M

meta information 16

method signatures 28

Multitier architecture

defined 2

multitier architecture

example 3

P

persistent data 28

PortableRemoteObject class 7

primary key 28

duplicate 28

R

remote interface 12

request object 7

response object 7

run application 25, 60, 70, 87

S

ServletException class 7

session Bean

defined 10

SessionBean interface 12

setSessionContext method 12

signatures, methods 28

T

thin-client application defined 2

transaction management 30

transaction rollback 28

U

Uninstall button 15

V

verify application 23, 58, 68, 86

W

Web Archive (WAR) file 19