11

Case Study: Servlet and JSP Bookstore

Objectives

- To build a three-tier, client/server, distributed Web application using Java servlet and JavaServer Pages technology.
- To be able to perform servlet/JSP interactions.
- To be able to use a **`RequestDispatcher`** to forward requests to another resource for further processing.
- To be able to create XML from a servlet and XSL transformations to convert the XML into a format the client can display.
- To introduce the Java 2 Enterprise Edition reference implementation server.
- To be able to deploy a Web application using the Java 2 Enterprise Edition.

*[*** NEED QUOTES. ***]*

advjhtp1_11.fm Page 528 Saturday, June 9, 2001 11:04 AM

Outline

11.1 Introduction
11.2 Bookstore Architecture
11.3 Entering the Bookstore
11.4 Obtaining the Book List from the Database
11.5 Viewing a Book's Details
11.6 Adding an Item to the Shopping Cart
11.7 Viewing the Shopping Cart
11.8 Checking Out
11.9 Processing the Order
11.10 Deploying the Bookstore Application in J2EE 1.2.1
11.10.1 Configuring the `books` Data Source
11.10.2 Launching the Cloudscape Database and J2EE Servers
11.10.3 Launching the J2EE Application Deployment Tool
11.10.4 Creating the Bookstore Application and Adding Library JARs
11.10.5 Creating `BookServlet` and `AddToCartServlet` Web Components
11.10.6 Adding Non-Servlet Components to the Application
11.10.7 Specifying the Web Context, Resource References, JNDI Names and Welcome Files
11.10.8 Deploying and Executing the Application

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

11.1 Introduction

This chapter serves as a capstone for our presentation of JSP and servlets. Here, we implement a bookstore Web application that integrates JDBC, XML, JSP and servlet technologies. The case study introduces additional servlet features that are discussed as they are encountered in the case study.

This chapter also serves as an introduction to the *Java 2 Enterprise Edition 1.2.1 reference implementation* used in Chapters 14–18. Unlike the JSP and servlet chapters that demonstrated examples using Apache's Tomcat JSP and servlet container, this chapter deploys the bookstore application on the J2EE 1.2.1 reference implementation application server software that is downloadable from **`java.sun.com/j2ee/download.html`** (see Appendix G for installation instructions). The J2EE 1.2.1 reference implementation includes the Apache Tomcat JSP and servlet container. After reading this chapter, you will be able to implement a substantial distributed Web application with many components and you will be able to deploy that application on the J2EE 1.2.1 application server.

advjhtp1_11.fm Page 529 Saturday, June 9, 2001 11:04 AM

11.2 Bookstore Architecture

This section overviews the architecture of the **Bug2Bug.com** bookstore application. We present a diagram of the basic interactions between XHTML documents, JSPs and servlets. Also, we present a table of all the documents and classes used in the case study. Our sample outputs demonstrate how the XHTML documents sent to the client are rendered.

Our **Bug2Bug.com** shopping cart case study consists of a series of XHTML documents, JSPs and servlets that interact to simulate a bookstore selling Deitel publications. This case study is implemented as a distributed, three-tier, Web-based application. The client tier is represented by the user's Web browser. The browser displays static XHTML documents and dynamically created XHTML documents that allow the user to interact with the server tier. The server tier consists of several JSPs and servlets that act on behalf of the client. These JSPs and servlets perform tasks such as creating a list of publications, creating documents containing the details about a publication, adding items to the shopping cart, viewing the shopping cart and processing the final order. Some of the JSPs and servlets perform database interactions on behalf of the client.

The database tier uses the **books** database introduced in Chapter 8, Java Database Connectivity. In this case study, we use only the **titles** table from the database (see Chapter 8).

Figure 11.1 illustrates the interactions between the bookstore's application components. In the diagram, names without file extensions (**displayBook** and **addToCart**) represent servlet aliases (i.e., the names used to invoke the servlets). As you will see when we deploy the case study in Section 11.10, the Java 2 Enterprise Edition 1.2.1 reference implementation includes an *Application Deployment Tool*. Among its many features, this tool enables us to specify the alias used to invoke a servlet. For example, **addToCart** is the alias for servlet **AddToCartServlet**. The Application Deployment Tool creates the deployment descriptor for a servlet as part of deploying an application.

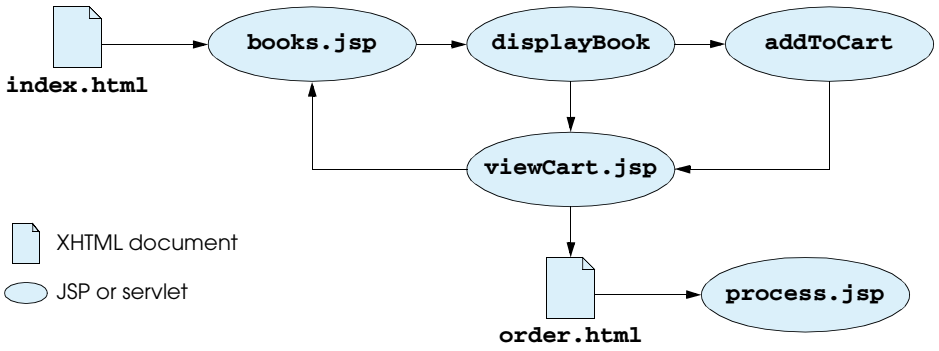

Fig. 11.1 **Bug2Bug.com** bookstore component interactions.

After the application is deployed, users can visit the bookstore by entering the following URL in a browser:

```
http://localhost:8000/advjhtp1/store
```

advjhtp1_11.fm Page 530 Saturday, June 9, 2001 11:04 AM

This URL requests the default home page for the store (**index.html**). The user can view the list of products by clicking a button on the home page. This invokes **books.jsp** which interacts with a database to create the list of books dynamically. The result is an XHTML document containing links to the servlet with alias **displayBook**. This servlet receives as a parameter the ISBN number for the selected book and returns an XHTML document containing the information for that book. From this document, the user can click buttons to place the current book in the shopping cart or view the shopping cart. Adding a book to a shopping cart invokes the servlet with alias **addToCart**. Viewing the cart contents invokes **viewCart.jsp** to return an XHTML document containing the cart contents, subtotals the dollar cost of each item and a total the dollar cost of all the items in the cart. When the user adds an item to the shopping cart, the **addToCart** servlet processes the user's request, then forwards it to **viewCart.jsp** to create the document that displays the current cart. At this point the user can either continue shopping (**books.jsp**) or proceed to checkout (**order.html**). In the latter case, the user is presented with a form to input name, address and credit-card information. Then, the user submits the form to invoke **process.jsp**, which completes the transaction by sending a confirmation document to the user.

Figure 11.2 overviews the XHTML documents, JSPs, servlets, JavaBeans and other files used in this case study.

File	Description
index.html	This is the default home page for the bookstore, which is displayed by entering the following URL in the client's Web browser: **http://localhost:8000/advjhtp1/store**
styles.css	This Cascading Style Sheet (CSS) file is linked to all XHTML documents rendered on the client. The CSS file allows us to apply uniform formatting across all the XHTML static and dynamic documents rendered.
books.jsp	This JSP uses **BookBean** objects and a **TitlesBean** object to create an XHTML document containing the product list. The **TitlesBean** object queries the **books** database to obtain the list of titles in the database. The results are processed and placed in an **ArrayList** of **BookBean** objects. The list is stored as a session attribute for the client.
BookBean.java	An instance of this JavaBean represents the data for one book. The bean's **getXML** method returns an XML **Element** representing the book.
TitlesBean.java	JSP **books.jsp** uses an instance of this **JavaBean** to obtain an **ArrayList** containing a **BookBean** for every product in the database.

Fig. 11.2 Servlet and JSP components for bookstore case study.

advjhtp1_11.fm Page 531 Saturday, June 9, 2001 11:04 AM

File	Description
`BookServlet.java`	This servlet (aliased as **`displayBook`** in Fig. 11.1) obtains the XML representation of a book selected by the user, then applies an XSL transformation to the XML to produce an XHTML document that can be rendered by the client. In this example, the client is assumed to be a browser that supports Cascading Style Sheets (CSS). Later examples in this book apply different XSL transformations for different client types.
`book.xsl`	This XSL style sheet specifies how to transform the XML representation of a book into an XHTML document that the client browser can render.
`CartItemBean.java`	An instance of this JavaBean maintains a **`BookBean`** and the current quantity for that book in the shopping cart. These beans are stored in a **`HashMap`** that represents the shopping cart contents.
`AddToCartServlet.java`	This servlet (aliased as **`addToCart`** in Fig. 11.1) updates the shopping cart. If the cart does not exist, the servlet creates a cart (a **`HashMap`** in this example). If a **`CartItemBean`** for the item is already in the cart, the servlet updates the quantity of that item in the bean. Otherwise, the servlet creates a new **`CartItemBean`** with a quantity of 1. After updating the cart, the user is forwarded to **`viewCart.jsp`** to view the current cart contents.
`viewCart.jsp`	This JSP extracts the **`CartItemBean`**s from the shopping cart, subtotals each item in the cart, totals all the items in the cart and creates an XHTML document that allows the client to view the cart in tabular form.
`order.html`	When viewing the cart, the user can click a **Check Out** button to view this order form. In this example, the form has no functionality. However, it is provided to help complete the application.
`process.jsp`	This final JSP pretends to process the user's credit-card information and creates an XHTML document indicating that the order was processed and the total order value.

Fig. 11.2 Servlet and JSP components for bookstore case study.

11.3 Entering the Bookstore

Figure 11.3 (**`index.html`**) is the default home page for the **`Bug2Bug.com`** bookstore. This is also known as the *welcome file*—an option specified at application deployment time (see Section 11.10). When the bookstore application is running on your computer in the Java 2 Enterprise Edition 1.2.1 reference implementation, you can enter the following URL in your Web browser to display the home page:

```
http://localhost:8000/advjhtp1/store
```

advjhtp1_11.fm Page 532 Saturday, June 9, 2001 11:04 AM


```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "DTD/xhtml1-strict.dtd">
4  <!-- index.html -->
5
6  <html xmlns = "http://www.w3.org/1999/xhtml"
7     xml:lang = "en" lang = "en">
8
9  <head>
10    <title>Shopping Cart Case Study</title>
11
12    <link rel = "stylesheet" href = "styles.css"
13       type = "text/css" />
14 </head>
15
16 <body>
17    <p class = "bigFont">Bug2Bug.com</p>
18
19    <p class = "bigFont italic">
20       Deitel & Associates, Inc.<br />
21       Shopping Cart Case Study
22    </p>
23
24    <!-- form to request books.jsp -->
25    <form method = "post" action = "books.jsp">
26       <input type = "submit" name = "enterButton"
27          value = "Click here to enter store" />
28    </form>
29 </body>
30
31 </html>
```

Fig. 11.3 Bookstore home page (**index.html**).

advjhtp1_11.fm Page 533 Saturday, June 9, 2001 11:04 AM

Lines 12–13 specify a linked style sheet **styles.css** (Fig. 11.4). All XHTML documents sent to the client use this style sheet, so that uniform formatting can be applied to the documents. The **form** at lines 25–28 provides a **submit** button that enables you to enter the store. Pressing this button invokes **books.jsp** (Fig. 11.7), which creates and returns an XHTML document containing the product list.

Figure 11.4 (**styles.css**) defines the common styles for rendering XHTML documents in this case study. Lines 1–2 indicate that all text in the **body** element should be centered and that the background color of the body should be steel blue. The background color is represented by the hexadecimal number **#b0c4de**. Line 3 defines class **.bold** to apply bold font weight to text. Lines 4–7 define class **.bigFont** with four CSS attributes. Elements to which this class is applied appear in the bold, Helvetica font which is double the size of the base-text font. The color of the font is dark blue (represented by the hexadecimal number **#00008b**). If Helvetica font is not available, the browser will attempt to use Arial, then the generic font **sans-serif** as a last resort. Line 8 defines class **.italic** to apply italic font style to text. Line 9 defines class **.right** to right justify text. Lines 10–11 indicate that all **table**, **th** (table head data) and **td** (table data) elements should have a three-pixel, grooved border with five pixels of internal padding between the text in a table cell and the border of that cell. Lines 12–14 indicate that all **table** elements should have bright blue background color (represented by the hexadecimal number **#6495ed**), and that all **table** elements should use automatically determined margins on both their left and right sides. This causes the table to be centered on the page. Not all of these styles are used in every XHTML document. However, using a single linked style sheet allows us to change the look and feel of our store quickly and easily by modifying the CSS file. For more information on CSS visit

```
www.w3.org/Style/CSS
```

Here you will find the CSS specifications. Wach specification includes an index of all the current CSS attributes and their values.

Portability Tip 11.1

Different browsers have different levels of support for Cascading Style Sheets.

```
 1  body            { text-align: center;
 2                    background-color: #boc4de }
 3  .bold           { font-weight: bold }
 4  .bigFont        { font-family: helvetica, arial, sans-serif;
 5                    font-weight: bold;
 6                    font-size: 2em;
 7                    color: #00008b }
 8  .italic         { font-style: italic }
 9  .right          { text-align: right }
10  table, th, td   { border: 3px groove;
11                    padding: 5px }
```

Fig. 11.4 Shared cascading style sheet (**styles.css**) used to apply common formatting across XHTML documents rendered on the client.

advjhtp1_11.fm Page 534 Saturday, June 9, 2001 11:04 AM


```
12  table          { background-color: #6495ed;
13                   margin-left: auto;
14                   margin-right: auto }
```

Fig. 11.4 Shared cascading style sheet (**styles.css**) used to apply common formatting across XHTML documents rendered on the client.

11.4 Obtaining the Book List from the Database

JavaServer Pages often generate XHTML that is sent to the client for rendering. JSP **books.jsp** (Fig. 11.7) generates an XHTML document containing a list of hyperlinks to information about each book in the **titles** table of the **books** database. From this list, the user can view information about a particular book by clicking the hyperlink for that book. This JSP uses a **TitlesBean** (Fig. 11.5) object and **BookBean** (Fig. 11.6) objects to create the product list. Each of the JavaBeans and **books.jsp** are discussed in this section. Figure 11.7 shows the rendering of the XHTML document sent to the browser by **books.jsp**.

The **TitlesBean** (Fig. 11.5) JavaBean performs a database query to obtain the list of titles in the database. Then, the results are processed and placed in an **ArrayList** of **BookBean** objects. As we will see in Fig. 11.7, **ArrayList** is stored by **books.jsp** as a session attribute for the client.

```
1  // TitlesBean.java
2  // Class TitlesBean makes a database connection and retrieves
3  // the books from the database.
4  package com.deitel.advjhtp1.store;
5
6  // Java core packages
7  import java.io.*;
8  import java.sql.*;
9  import java.util.*;
10
11 // Java extension packages
12 import javax.naming.*;
13 import javax.sql.*;
14
15 public class TitlesBean implements Serializable {
16    private Connection connection;
17    private PreparedStatement titlesQuery;
18
19    // construct TitlesBean object
20    public TitlesBean()
21    {
22       // attempt database connection and setup SQL statements
23       try {
24          InitialContext ic = new InitialContext();
25
26          DataSource source =
```

Fig. 11.5 **TitlesBean** for obtaining book information from the books database and creating an **ArrayList** of **BookBean** objects.

advjhtp1_11.fm Page 535 Saturday, June 9, 2001 11:04 AM


```
            ( DataSource ) ic.lookup(
               "java:comp/env/jdbc/books" );

         connection = source.getConnection();

         titlesQuery =
            connection.prepareStatement(
               "SELECT isbn, title, editionNumber, " +
               "copyRight, publisherID, imageFile, price " +
               "FROM titles ORDER BY title"
            );
      }

      // process exceptions during database setup
      catch ( SQLException sqlException ) {
         sqlException.printStackTrace();
      }

      // process problems locating data source
      catch ( NamingException namingException ) {
         namingException.printStackTrace();
      }
   }

   // return an ArrayList of BookBeans
   public ArrayList getTitles()
   {
      ArrayList titlesList = new ArrayList();

      // obtain list of titles
      try {
         ResultSet results = titlesQuery.executeQuery();

         // get row data
         while ( results.next() ) {
            BookBean book = new BookBean();

            book.setISBN( results.getString( "isbn" ) );
            book.setTitle( results.getString( "title" ) );
            book.setEditionNumber(
               results.getInt( "editionNumber" ) );
            book.setCopyright( results.getString( "copyright" ) );
            book.setPublisherID(
               results.getInt( "publisherID" ) );
            book.setImageFile( results.getString( "imageFile" ) );
            book.setPrice( results.getDouble( "price" ) );

            titlesList.add( book );
         }
      }

      // process exceptions during database query
      catch ( SQLException exception ) {
```

Fig. 11.5 **`TitlesBean`** for obtaining book information from the books database and creating an **`ArrayList`** of **`BookBean`** objects.

advjhtp1_11.fm Page 536 Saturday, June 9, 2001 11:04 AM


```
80             exception.printStackTrace();
81          }
82
83          // return the list of titles
84          finally {
85             return titlesList;
86          }
87       }
88
89       // close statements and terminate database connection
90       protected void finalize()
91       {
92          // attempt to close database connection
93          try {
94             connection.close();
95          }
96
97          // process SQLException on close operation
98          catch ( SQLException sqlException ) {
99             sqlException.printStackTrace();
100         }
101      }
102   }
```

Fig. 11.5 **`TitlesBean`** for obtaining book information from the books database and creating an **`ArrayList`** of **`BookBean`** objects.

The JavaBean **`TitlesBean`** requires us to introduce the *Java Naming and Directory Interface (JNDI)*. Enterprise Java applications often access information and resources (such as databases) that are external to those applications. In some cases, those resources are distributed across a network. Just as an RMI client uses the RMI registry to locate a server object so the client can request a server, Enterprise application components must be able to locate the resources they use. An Enterprise Java application container must provide a *naming service* that implements JNDI and enables the components executing in that container to perform name lookups to locate resources. The J2EE 1.2.1 reference implementation server includes such a naming service that we use to locate our **`books`** database at execution time.

The **`TitlesBean`** uses JNDI to interact with the naming service and locate the data source (i.e., the **`books`** database). The **`TitlesBean`** constructor (lines 20–49) attempts the connection to the database using class **`InitialContext`** from package **`javax.naming`** and interface ***`DataSource`*** from package **`javax.sql`**. When you deploy an Enterprise Java application (Section 11.10), you specify the resources (such as databases) used by the applicationand the JNDI names for those resources. Using an **`InitialContext`**, an Enterprise application component can look up a resource. The **`InitialContext`** provides access to the application's *naming environment*.

Line 24 creates a new **`InitialContext`**. The **`InitialContext`** constructor throws a **`NamingException`** if it cannot locate a naming service. Lines 26–28 invoke **`InitialContext`** method **`lookup`** to locate our **`books`** data source. In the argument, the text **`java:comp/env`** indicates that method **`lookup`** should search for the resource in the application's component environment entries (i.e., the resource names specified at

advjhtp1_11.fm Page 537 Saturday, June 9, 2001 11:04 AM

deployment time). The text **jdbc/books** indicates that the resource is a JDBC data source called **books**. Method **lookup** returns a **DataSource** object and throws a **NamingException** if it cannot resolve the name it receives as an argument. Line 25 uses the **DataSource** to connect to the database. Lines 32–37 create a **PreparedStatement** that, when executed, returns the information about each title from the **titles** table of the **books** database.

Method **getTitles** (lines 52–87) returns an **ArrayList** (**titlesList**) containing a **BookBean** JavaBean for each title in the database. Line 58 executes **titlesQuery**. Lines 57–76 process the **ResultSet** (**results**). For each row in **results**, line 62 creates a new **BookBean** and lines 64–72 set the attributes of the **BookBean** to columns in that **ResultSet** row. **ResultSet** methods **getString**, **getInt** and **getDouble** return the column data in the appropriate formats. Line 74 adds the new **BookBean** to **titlesList**. In the **finally** block, **titlesList** is returned. If there is an exception while performing the database interactions or there are no records in the database, the **ArrayList** will be empty.

An instance of the **BookBean** JavaBean represents the properties for one book, including the book's ISBN, title, copyright, cover image file name, edition number, publisher ID number and price. Each of these is a read/write property. Some of this information is not used in this example. **BookBean** method **getXML** returns an XML **Element** representing the book.

```
1  // BookBean.java
2  // A BookBean object contains the data for one book.
3  package com.deitel.advjhtp1.store;
4
5  // Java core packages
6  import java.io.*;
7  import java.text.*;
8  import java.util.*;
9
10 // third-party packages
11 import org.w3c.dom.*;
12
13 public class BookBean implements Serializable {
14    private String ISBN, title, copyright, imageFile;
15    private int editionNumber, publisherID;
16    private double price;
17
18    // set ISBN number
19    public void setISBN( String isbn )
20    {
21       ISBN = isbn;
22    }
23
24    // return ISBN number
25    public String getISBN()
26    {
```

Fig. 11.6 **BookBean** that represents a single book's information and defines the XML format of that information.

advjhtp1_11.fm Page 538 Saturday, June 9, 2001 11:04 AM


```
27          return ISBN;
28       }
29
30       // set book title
31       public void setTitle( String bookTitle )
32       {
33          title = bookTitle;
34       }
35
36       // return book title
37       public String getTitle()
38       {
39          return title;
40       }
41
42       // set copyright year
43       public void setCopyright( String year )
44       {
45          copyright = year;
46       }
47
48       // return copyright year
49       public String getCopyright()
50       {
51          return copyright;
52       }
53
54       // set file name of image representing product cover
55       public void setImageFile( String fileName )
56       {
57          imageFile = fileName;
58       }
59
60       // return file name of image representing product cover
61       public String getImageFile()
62       {
63          return imageFile;
64       }
65
66       // set edition number
67       public void setEditionNumber( int edition )
68       {
69          editionNumber = edition;
70       }
71
72       // return edition number
73       public int getEditionNumber()
74       {
75          return editionNumber;
76       }
77
78       // set publisher ID number
79       public void setPublisherID( int id )
```

Fig. 11.6 **BookBean** that represents a single book's information and defines the XML format of that information.

advjhtp1_11.fm Page 539 Saturday, June 9, 2001 11:04 AM


```
80       {
81          publisherID = id;
82       }
83
84       // return publisher ID number
85       public int getPublisherID()
86       {
87          return publisherID;
88       }
89
90       // set price
91       public void setPrice( double amount )
92       {
93          price = amount;
94       }
95
96       // return price
97       public double getPrice()
98       {
99          return price;
100      }
101
102      // get an XML representation of the Product
103      public Element getXML( Document document )
104      {
105         // create product root element
106         Element product = document.createElement( "product" );
107
108         // create isbn element, append as child of product
109         Element temp = document.createElement( "isbn" );
110         temp.appendChild( document.createTextNode( getISBN() ) );
111         product.appendChild( temp );
112
113         // create title element, append as child of product
114         temp = document.createElement( "title" );
115         temp.appendChild( document.createTextNode( getTitle() ) );
116         product.appendChild( temp );
117
118         // create a currency formatting object for US dollars
119         NumberFormat priceFormatter =
120            NumberFormat.getCurrencyInstance( Locale.US );
121
122         // create price element, append as child of product
123         temp = document.createElement( "price" );
124         temp.appendChild( document.createTextNode(
125            priceFormatter.format( getPrice() ) ) );
126         product.appendChild( temp );
127
128         // create imageFile element, append as child of product
129         temp = document.createElement( "imageFile" );
130         temp.appendChild(
131            document.createTextNode( getImageFile() ) );
132         product.appendChild( temp );
```

Fig. 11.6 **BookBean** that represents a single book's information and defines the XML format of that information.

advjhtp1_11.fm Page 540 Saturday, June 9, 2001 11:04 AM


```
133
134         // create copyright element, append as child of product
135         temp = document.createElement( "copyright" );
136         temp.appendChild(
137            document.createTextNode( getCopyright() ) );
138         product.appendChild( temp );
139
140         // create publisherID element, append as child of product
141         temp = document.createElement( "publisherID" );
142         temp.appendChild( document.createTextNode(
143            String.valueOf( getPublisherID() ) ) );
144         product.appendChild( temp );
145
146         // create editionNumber element, append as child of product
147         temp = document.createElement( "editionNumber" );
148         temp.appendChild( document.createTextNode(
149             String.valueOf( getEditionNumber() ) ) );
150         product.appendChild( temp );
151
152         // return product element
153         return product;
154      }
155   }
```

Fig. 11.6 **BookBean** that represents a single book's information and defines the XML format of that information.

Method **getXML** (lines 103–154) uses the **org.w3c.dom** package's **Document** and **Element** interfaces to create an XML representation of the book data as part of the **Document** that is passed to the method as an argument. The complete information for one book is placed in a **product** element (created at line 106). The elements for the individual properties of a book are appended to the **product** element as children. For example, line 109 uses **Document** method **createElement** to create element **isbn**. Line 110 uses **Document** method **createTextNode** to specify the text in the **isbn** element, and uses **Element** method **appendChild** to append the text to element **isbn**. Then, line 111 appends element **isbn** as a child of element **product** with **Element** method **appendChild**. Similar operations are performed for the other book properties. Lines 119–120obtain a **NumberFormat** object that formats currency for the United States locale. This is used to format the book price US dollars (line 125). Line 150 returns element **product** to the caller. We revisit method **getXML** in our **BookServlet** discussion (Fig. 11.8). For more information about XML and Java, refer to Appendices A–F.

JavaServer Page **books.jsp** dynamically generates the list of titles as an XHTML document to be rendered on the client. Lines 7–11 specify the JSP page settings. This JSP uses classes from our store package (**com.deitel.advjhtp1.store**) and package **java.util**. Also, this JSP uses session-tracking features. The dynamic parts of this JSP are defined in lines 30–64 with JSP scriptlets and expressions.

advjhtp1_11.fm Page 541 Saturday, June 9, 2001 11:04 AM


```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "DTD/xhtml1-strict.dtd">
4  <!-- books.jsp -->
5
6  <%-- JSP page settings --%>
7  <%@
8     page language = "java"
9     import = "com.deitel.advjhtp1.store.*, java.util.*"
10    session = "true"
11 %>
12
13 <!-- begin document -->
14 <html xmlns = "http://www.w3.org/1999/xhtml"
15    xml:lang = "en" lang = "en">
16
17 <head>
18    <title>Book List</title>
19
20    <link rel = "stylesheet" href = "styles.css"
21       type = "text/css" />
22 </head>
23
24 <body>
25    <p class = "bigFont">Available Books</p>
26
27    <p class = "bold">Click a link to view book information</p>
28
29    <%-- begin JSP scriptlet to create list of books --%>
30    <%
31       TitlesBean titlesBean = new TitlesBean();
32       ArrayList titles = titlesBean.getTitles();
33       BookBean currentBook;
34
35       // store titles in session for further use
36       session.setAttribute( "titles", titles );
37
38       // obtain an Iterator to the set of keys in the HashMap
39       Iterator iterator = titles.iterator();
40
41       // use the Iterator to get each BookBean and create
42       // a link to each book
43       while ( iterator.hasNext() ) {
44          currentBook = ( BookBean ) iterator.next();
45
46    %> <%-- end scriptlet to insert literal XHTML and --%>
47       <%-- JSP expressions output from this loop     --%>
48
49          <%-- link to a book's information --%>
50          <span class = "bold">
```

Fig. 11.7 JSP **books.jsp** returns to the client an XHTML document containing the book list.

advjhtp1_11.fm Page 542 Saturday, June 9, 2001 11:04 AM


```
51                <a href =
52                   "displayBook?isbn=<%= currentBook.getISBN() %>">
53
54                   <%= currentBook.getTitle() + ", " +
55                      currentBook.getEditionNumber() + "e" %>
56                </a>
57                <br />
58             </span>
59
60       <% // continue scriptlet
61
62          }   // end while loop
63
64       %> <%-- end scriptlet --%>
65    </body>
66
67    </html>
```

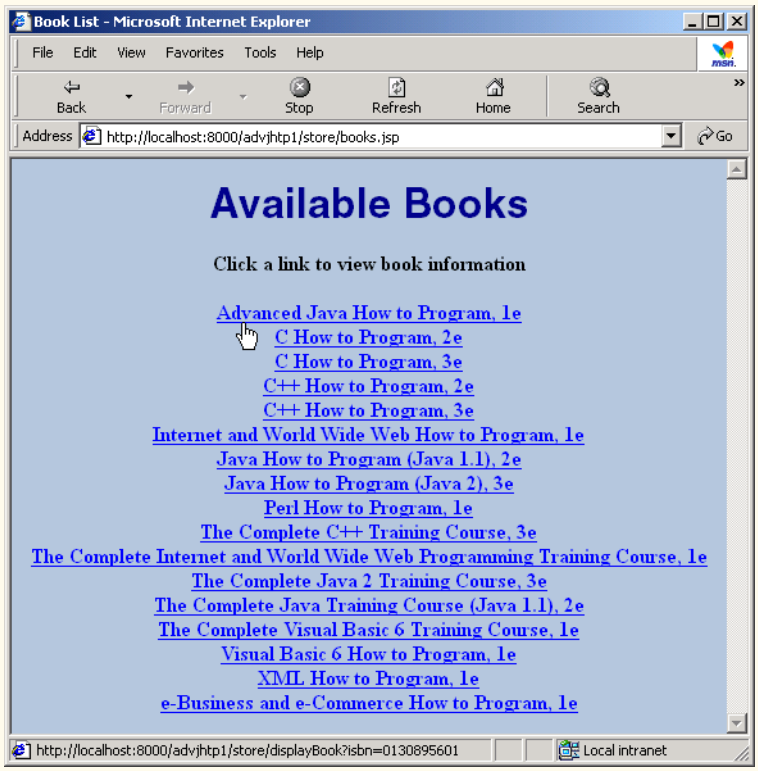

Fig. 11.7 JSP **books.jsp** returns to the client an XHTML document containing the book list.

The scriptlet begins at line 30. Line 31 creates a **TitlesBean** and line 32 invokes its **getTitles** method to obtain the **ArrayList** of **BookBean** objects. Line 36 sets a **titles** session attribute to store the **ArrayList** for use later in the client's session. Line

advjhtp1_11.fm Page 543 Saturday, June 9, 2001 11:04 AM

39 obtains an **Iterator** for the **ArrayList**. Lines 43–44 begin a loop that uses the **Iterator** to output each hyperlink. The scriptlet temporarily terminates here so that lines 50–58 can insert XHTML markup. In this markup, line 52 uses a JSP expression to insert the book's ISBN number as the value in a name/value pair that is passed to the **displayBook** servlet (**BookServlet**) as an argument. Lines 54–55 use another JSP expression to insert the book's title and edition number as the text displayed for the hyperlink. Lines 60–64 continue the scriptlet with the closing curly brace of the **while** loop that started at line 43.

11.5 Viewing a Book's Details

Like many companies, **Bug2Bug.com** is beginning to use XML on its Web site. When the user selects a book in **books.jsp**, the **Bug2Bug.com** converts the book information to XML. **BookSerlvet** (Fig. 11.8) transforms the XML representation of the book into an XHTML document using XSL style sheet **book.xsl** (Fig. 11.9).

```
1  // BookServlet.java
2  // Servlet to return one book's information to the client.
3  // The servlet produces XML which is transformed with XSL to
4  // produce the client XHTML page.
5  package com.deitel.advjhtp1.store;
6
7  // Java core packages
8  import java.io.*;
9  import java.util.*;
10
11 // Java extension packages
12 import javax.servlet.*;
13 import javax.servlet.http.*;
14 import javax.xml.parsers.*;
15 import javax.xml.transform.*;
16 import javax.xml.transform.dom.*;
17 import javax.xml.transform.stream.*;
18
19 // third-party packages
20 import org.w3c.dom.*;
21 import org.xml.sax.*;
22
23 public class BookServlet extends HttpServlet {
24    protected void doGet( HttpServletRequest request,
25       HttpServletResponse response )
26       throws ServletException, IOException
27    {
28       HttpSession session = request.getSession( false );
29
30       // RequestDispatcher to forward client to bookstore home
31       // page if no session exists or no books are selected
```

Fig. 11.8 **BookServlet** obtains the XML representation of a book and applies an XSL transformation to output an XHTML document as the response to the client.

advjhtp1_11.fm Page 544 Saturday, June 9, 2001 11:04 AM


```
32          RequestDispatcher dispatcher =
33             request.getRequestDispatcher( "/index.html" );
34
35          // if session does not exist, forward to index.html
36          if ( session == null )
37             dispatcher.forward( request, response );
38
39          // get books from session object
40          ArrayList titles =
41             ( ArrayList ) session.getAttribute( "titles" );
42
43          // locate BookBean object for selected book
44          Iterator iterator = titles.iterator();
45          BookBean book = null;
46
47          String isbn = request.getParameter( "isbn" );
48
49          while ( iterator.hasNext() ) {
50             book = ( BookBean ) iterator.next();
51
52             if ( isbn.equals( book.getISBN() ) ) {
53
54                // save the book in a session attribute
55                session.setAttribute( "bookToAdd", book );
56                break;  // isbn matches current book
57             }
58          }
59
60          // if book is not in list, forward to index.html
61          if ( book == null )
62             dispatcher.forward( request, response );
63
64          // get XML document and transform for browser client
65          try {
66             // get a DocumentBuilderFactory for creating
67             // a DocumentBuilder (i.e., an XML parser)
68             DocumentBuilderFactory factory =
69                DocumentBuilderFactory.newInstance();
70
71             // get a DocumentBuilder for building the DOM tree
72             DocumentBuilder builder =
73                factory.newDocumentBuilder();
74
75             // create a new Document (empty DOM tree)
76             Document messageDocument = builder.newDocument();
77
78             // get XML from BookBean and append to Document
79             Element bookElement = book.getXML( messageDocument );
80             messageDocument.appendChild( bookElement );
81
82             // get PrintWriter for writing data to client
83             response.setContentType( "text/html" );
```

Fig. 11.8 **BookServlet** obtains the XML representation of a book and applies an XSL transformation to output an XHTML document as the response to the client.

advjhtp1_11.fm Page 545 Saturday, June 9, 2001 11:04 AM


```
84             PrintWriter out = response.getWriter();
85
86             // open InputStream for XSL document
87             InputStream xslStream =
88                getServletContext().getResourceAsStream(
89                   "/book.xsl" );
90
91             // transform XML document using XSLT
92             transform( messageDocument, xslStream, out );
93
94             // flush and close PrintWriter
95             out.flush();
96             out.close();
97          }
98
99          // catch XML parser exceptions
100         catch ( ParserConfigurationException pcException ) {
101            pcException.printStackTrace();
102         }
103      }
104
105      // transform XML document using provided XSLT InputStream
106      // and write resulting document to provided PrintWriter
107      public void transform( Document document,
108         InputStream xslStream, PrintWriter output )
109      {
110         try {
111            // create DOMSource for source XML document
112            DOMSource xmlSource = new DOMSource( document );
113
114            // create StreamSource for XSLT document
115            StreamSource xslSource =
116               new StreamSource( xslStream );
117
118            // create StreamResult for transformation result
119            StreamResult result = new StreamResult( output );
120
121            // create TransformerFactory to obtain a Transformer
122            TransformerFactory transformerFactory =
123               TransformerFactory.newInstance();
124
125           // create Transformer for performing XSL transformation
126            Transformer transformer =
127               transformerFactory.newTransformer( xslSource );
128
129           // perform transformation and deliver content to client
130            transformer.transform( xmlSource, result );
131         }
132
133         // handle exception when transforming XML document
134         catch ( TransformerException transformerException ) {
135            transformerException.printStackTrace( System.err );
```

Fig. 11.8 **`BookServlet`** obtains the XML representation of a book and applies an XSL transformation to output an XHTML document as the response to the client.

advjhtp1_11.fm Page 546 Saturday, June 9, 2001 11:04 AM


```
136         }
137      }
138   }
```

Fig. 11.8 **BookServlet** obtains the XML representation of a book and applies an XSL transformation to output an XHTML document as the response to the client.

There are two major parts in **BookServlet**'s **doGet** method (lines 24–103). Lines 28–62 locate the **BookBean** for the book selected by the user in **books.jsp**. Lines 65–102 process the XML representation of a book and apply an XSL transformation to that XML.

Line 28 obtains the **HttpSession** object for the current client. This object contains a session attribute indicating the book selected by the user in **books.jsp**. Lines 32–33 obtain a ***RequestDispatcher*** for the **"/index.html"** document by calling **ServletRequest** method **getRequestDispatcher**. A **RequestDispatcher** (package **javax.servlet**) provides two methods—***forward*** and ***include***—that enable a servlet to forward a client request to another resource or include content from another resource in a servlet's response. In this example, if there is no session object for the current client (lines 36–37) or if there is no book selected (lines 61–62), the request is forwarded back to the **index.html** home page of our bookstore. Methods **forward** and **include** each take two arguments—the **HttpServletRequest** and **HttpServletResponse** objects for the current request.

Software Engineering Observation 11.1

When ***RequestDispatcher*** *method* ***forward*** *is called, processing of the request by the current servlet terminates.*

Note that **RequestDispatcher** objects can be obtained with method **getRequestDispatcher** from an object that implements interface **ServletRequest** or from the **ServletContext** with methods **getRequestDispatcher** or **getNamedDispatcher**. **ServletContext** method **getNamedDispatcher** receives the name of a servlet as an argument, then searches the **ServletContext** for a servlet by that name. If no such servlet is found, this method returns **null**. Both the **ServletRequest** and the **ServletContext** **getRequestDispatcher** methods simply return the content of the specified path if the path does not represent a servlet.

Lines 40–41 get the **ArrayList** of **BookBean**s from the session object. Lines 44–58 perform a linear search to locate the **BookBean** for the selected book. The ISBN for that book is stored in an **isbn** parameter passed to the servlet (retrieved on line 47). If the **BookBean** is found, line 55 sets session attribute **bookToAdd** with that **BookBean** as the attribute's value. **AddToCartServlet** (Fig. 11.10) uses this attribute to update the shopping cart.

The **try** block (lines 65–97) performs the XML and XSL processing that results in an XHTML document containing a single book's information. Before the XML and XSL capabilities can be used, you must download and install Sun's *Java API for XML Parsing (JAXP)* version 1.1 from **java.sun.com/xml/download.htm**. The root directory of JAXP (**jaxp-1.1**) contains three JAR files **crimson.jar**, **jaxp.jar** and **xalan.jar** that are required for compiling and running programs that use JAXP. These

advjhtp1_11.fm Page 547 Saturday, June 9, 2001 11:04 AM

must be added to the Java extension mechanism for your Java 2 Standard Edition installation. Place a *copy* of these files in your Java installation's extensions directory (**jre/lib/ext** on Linux/UNIX and **jre\lib\ext** on Windows). When we deploy this application in the Java 2 Enterprise Edition 1.2.1 reference implementation server, we will discuss how to incorporate JAXP into the application.

Software Engineering Observation 11.2

JAXP 1.1 is part of the J2EE 1.3 reference implementation.

Creating a Document Object Model (DOM) tree from an XML document requires a ***DocumentBuilder*** parser object. **DocumentBuilder** objects are obtained from a ***DocumentBuilderFactory***. Lines 68–69 obtain a **DocumentBuilderFactory**, which lines 72–73 use to obtain a **DocumentBuilder** parser object. The **DocumentBuilder** enables the program to create a **Document** object tree in which the XML document elements are represented as **Element** objects. Line 76 uses the **DocumentBuilder** object builder to create a new **Document**. Line 79 invokes the **BookBean**'s **getXML** method to obtain an **Element** representation of the book. Line 80 appends this **Element** to **messageDocument** (the **Document** object). Classes **DocumentBuilderFactory** and **DocumentBuilder** are located in package **javax.xml.parsers**. Classes **Document** and **Element** are located in package **org.w3c.dom**. [Note: For detailed information on XML, see Appendices A–F.]

Next, line 83 specifies the response content type and line 84 obtains a **PrintWriter** to output the response to the client. Lines 87–89 create an **InputStream** that will be used by the XSL transformation processor to read the XSL file. The response is created by the XSL transformation performed in method **transform** (lines 107–137). We pass three arguments to this method—the XML **Document** to which the XSL transformation that will be applied (**messageDocument**), the **InputStream** that reads the XSL file (**xslStream**) and the target stream to which the results should be written (**out**). The output target can be one of several types including a character stream (i.e., the **response** object's **PrintWriter** in this example).

Line 112 creates a **DOMSource** that represents the XML document. This serves as the source of the XML that is transformed. Lines 115–116 create a **StreamSource** for the XSL file. This serves as the source of the XSL that transforms the **DOMSource**. Line 119 creates a **StreamResult** for the **PrintWriter** to which the results of the XSL transformation are written. Lines 122–123 create a **TransformerFactory** with **static** method **newInstance**. This object enables the program to obtain a **Transformer** object that applies the XSL transformation. Lines 126–127 create a **Transformer** using **TransformerFactory** method **newTransformer**, which receives a **StreamSource** argument representing the XSL (**xslSource** in this example). Line 130 invokes **Transformer** method **transform** to perform the XSL transformation on the given **DOMSource** object (**xmlSource**) and writes the result to the given **StreamResult** object (**result**). Lines 134–136 catch a **TransformerException** if a problem occurs when creating the **TransformerFactory**, creating the **Transformer** or performing the transformation.

Figure 11.9 contains the **book.xsl** style sheet file used in the XSL transformation. The values of six elements in the XML document are placed in the resulting XHTML doc-

advjhtp1_11.fm Page 548 Saturday, June 9, 2001 11:04 AM

ument. Lines 23 and 30 place the book's **title** in the document's **title** element and in a paragraph at the beginning of the document's **body** element, respectively. Line 36 specifies an **img** element in which the value of the **imageFile** element of an XML document specifies the name of the file representing the book's cover image. Line 37 specifies the **alt** attribute of the **img** element using the book's **title**. Lines 43, 51, 59 and 67 place the book's **price**, **isbn**, **editionNumber** and **copyright** in table cells. The resulting XHTML document is shown in the screen capture at the end of Fig. 11.9. For more details on XSL, refer to Appendix F.

```
1  <?xml version = "1.0"?>
2
3  <xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
4     version = "1.0">
5
6  <xsl:output method = "xml" omit-xml-declaration = "no"
7     indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"
8     doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
9
10 <!-- book.xsl                                          -->
11 <!-- XSL document that transforms XML into XHTML -->
12
13 <!-- specify the root of the XML document -->
14 <!-- that references this stylesheet      -->
15 <xsl:template match = "product">
16
17    <html xmlns = "http://www.w3.org/1999/xhtml"
18       xml:lang = "en" lang = "en">
19
20    <head>
21
22       <!-- obtain book title from JSP to place in title -->
23       <title><xsl:value-of select = "title"/></title>
24
25       <link rel = "stylesheet" href = "styles.css"
26          type = "text/css" />
27    </head>
28
29    <body>
30       <p class = "bigFont"><xsl:value-of select = "title"/></p>
31
32       <table>
33          <tr>
34             <!-- create table cell for product image -->
35             <td rowspan = "5">  <!-- cell spans 5 rows -->
36                <img src = "images/{ imageFile }" border = "1"
37                   alt = "{ title }" />
38             </td>
39
40             <!-- create table cells for price in row 1 -->
```

Fig. 11.9 XSL style sheet that transforms a book's XML representation into an XHTML document.

advjhtp1_11.fm Page 549 Saturday, June 9, 2001 11:04 AM


```
41             <td class = "bold">Price:</td>
42
43             <td><xsl:value-of select = "price"/></td>
44          </tr>
45
46          <tr>
47
48             <!-- create table cells for ISBN in row 2 -->
49             <td class = "bold">ISBN #:</td>
50
51             <td><xsl:value-of select = "isbn"/></td>
52          </tr>
53
54          <tr>
55
56             <!-- create table cells for edition in row 3 -->
57             <td class = "bold">Edition:</td>
58
59             <td><xsl:value-of select = "editionNumber"/></td>
60          </tr>
61
62          <tr>
63
64             <!-- create table cells for copyright in row 4 -->
65             <td class = "bold">Copyright:</td>
66
67             <td><xsl:value-of select = "copyright"/></td>
68          </tr>
69
70          <tr>
71
72             <!-- create Add to Cart button in row 5 -->
73             <td>
74                <form method = "post" action = "addToCart">
75                   <input type = "submit" value = "Add to Cart" />
76                </form>
77             </td>
78
79             <!-- create View Cart button in row 5 -->
80             <td>
81                <form method = "post" action = "viewCart.jsp">
82                   <input type = "submit" value = "View Cart" />
83                </form>
84             </td>
85          </tr>
86       </table>
87
88    </body>
89
90    </html>
91
92 </xsl:template>
93
```

Fig. 11.9 XSL style sheet that transforms a book's XML representation into an XHTML document.

advjhtp1_11.fm Page 550 Saturday, June 9, 2001 11:04 AM


```
94   </xsl:stylesheet>
```

Fig. 11.9 XSL style sheet that transforms a book's XML representation into an XHTML document.

11.6 Adding an Item to the Shopping Cart

When the user presses the **Add to Cart** button in the XHTML document produced by the last section, the **`AddToCartServlet`** (aliased as **`addToCart`**) updates the shopping cart. If the cart does not exist, the servlet creates a shopping cart (a **`HashMap`** in this example). Items in the shopping cart are represented with **`CartItemBean`** objects. An instance of this JavaBean maintains a **`BookBean`** and the current quantity for that book in the shopping cart. When the user adds an item to the cart, if that item already is represented in the cart with a **`CartItemBean`**, the quantity of that item is updated in the bean. Otherwise, the servlet creates a new **`CartItemBean`** with a quantity of 1. After updating the cart, the user is forwarded to **`viewCart.jsp`** to view the current cart contents.

Class **`CartItemBean`** (Fig. 11.10) stores a **`BookBean`** and a quantity for that book. It maintains the **`BookBean`** as a read-only property of the bean and the **`quantity`** as a read-write property of the bean.

```
1  // CartItemBean.java
```

Fig. 11.10 **`CartItemBean`**s contain a **`BookBean`** and the **`quantity`** of a book in the shopping cart.

advjhtp1_11.fm Page 551 Saturday, June 9, 2001 11:04 AM


```
2  // Class that maintains a book and its quantity.
3  package com.deitel.advjhtp1.store;
4
5  import java.io.*;
6
7  public class CartItemBean implements Serializable {
8     private BookBean book;
9     private int quantity;
10
11    // no-argument constructor
12    public CartItemBean() { }
13
14    // initialize a CartItemBean
15    public CartItemBean( BookBean bookToAdd, int number )
16    {
17       book = bookToAdd;
18       quantity = number;
19    }
20
21    // get the book (this is a read-only property)
22    public BookBean getBook()
23    {
24       return book;
25    }
26
27    // set the quantity
28    public void setQuantity( int number )
29    {
30       quantity = number;
31    }
32
33    // get the quantity
34    public int getQuantity()
35    {
36       return quantity;
37    }
38 }
```

Fig. 11.10 **CartItemBean**s contain a **BookBean** and the **quantity** of a book in the shopping cart.

Class **AddToCartServlet** is shown in Fig. 11.11. The **AddToCartServlet**'s **doPost** method obtains the **HttpSession** object for the current client (line 18). If a session does not exist for this client, a **RequestDispatcher** forwards the request to the bookstore home page **index.html** (lines 22–26). Otherwise, line 29 obtains the value of session attribute **cart**—the **HashMap** that represents the shopping cart. Lines 30–31 obtain the value of session attribute **bookToAdd**—the **BookBean** representing the book to add to the shopping cart. If the shopping cart does not exist, lines 34–35 create a new **HashMap** to store the cart contents. Lines 38–39 attempt to locate the **CartItemBean** for the book being added to the cart. If one exists, line 44 increments the quantity for that **CartItemBean**. Otherwise, line 46 creates a new **CartItemBean** with a quantity of 1 and put it into the shopping cart (**HashMap cart**). Line 49 sets the **cart** session attribute to the **HashMap cart**. Then lines 52–54 create a **RequestDispatcher** for JSP

advjhtp1_11.fm Page 552 Saturday, June 9, 2001 11:04 AM

viewCart.jsp and **forward** the processing of the request to that JSP, so it can display the cart contents.

```
1  // AddToCartServlet.java
2  // Servlet to add a book to the shopping cart.
3  package com.deitel.advjhtp1.store;
4
5  // Java core packages
6  import java.io.*;
7  import java.util.*;
8
9  // Java extension packages
10 import javax.servlet.*;
11 import javax.servlet.http.*;
12
13 public class AddToCartServlet extends HttpServlet {
14    protected void doPost( HttpServletRequest request,
15       HttpServletResponse response )
16       throws ServletException, IOException
17    {
18       HttpSession session = request.getSession( false );
19       RequestDispatcher dispatcher;
20
21       // if session does not exist, forward to index.html
22       if ( session == null ) {
23          dispatcher =
24             request.getRequestDispatcher( "/index.html" );
25          dispatcher.forward( request, response );
26       }
27
28       // session exists, get cart HashMap and book to add
29       HashMap cart = ( HashMap ) session.getAttribute( "cart" );
30       BookBean book =
31          ( BookBean ) session.getAttribute( "bookToAdd" );
32
33       // if cart does not exist, create it
34       if ( cart == null )
35          cart = new HashMap();
36
37       // determine if book is in cart
38       CartItemBean cartItem =
39          ( CartItemBean ) cart.get( book.getISBN() );
40
41       // If book is already in cart, update its quantity.
42       // Otherwise, create an entry in the cart.
43       if ( cartItem != null )
44          cartItem.setQuantity( cartItem.getQuantity() + 1 );
45       else
46          cart.put( book.getISBN(), new CartItemBean( book, 1 ) );
47
48       // update the cart attribute
```

Fig. 11.11 **AddToCartServlet** places an item in the shopping cart and invokes **viewCart.jsp** to display the cart contents.

advjhtp1_11.fm Page 553 Saturday, June 9, 2001 11:04 AM


```
49          session.setAttribute( "cart", cart );
50
51          // send the user to viewCart.jsp
52          dispatcher =
53             request.getRequestDispatcher( "/viewCart.jsp" );
54          dispatcher.forward( request, response );
55       }
56    }
```

Fig. 11.11 **AddToCartServlet** places an item in the shopping cart and invokes **viewCart.jsp** to display the cart contents.

11.7 Viewing the Shopping Cart

JSP **viewCart.jsp** extracts the **CartItemBean**s from the shopping cart, subtotals each item in the cart, totals all the items in the cart and creates an XHTML document that allows the client to view the cart in tabular format. This JSP uses classes from our bookstore package (**com.deitel.advjhtp1.store**) and from packages **java.util** and **java.text**.

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "DTD/xhtml1-strict.dtd">
4  <!-- viewCart.jsp -->
5
6  <%-- JSP page settings --%>
7  <%@ page language = "java" session = "true" %>
8  <%@ page import = "com.deitel.advjhtp1.store.*" %>
9  <%@ page import = "java.util.*" %>
10 <%@ page import = "java.text.*" %>
11
12 <html xmlns = "http://www.w3.org/1999/xhtml"
13    xml:lang = "en" lang = "en">
14
15 <head>
16    <title>Shopping Cart</title>
17
18    <link rel = "stylesheet" href = "styles.css"
19       type = "text/css" />
20 </head>
21
22 <body>
23    <p class = "bigFont">Shopping Cart</p>
24
25 <%-- start scriptlet to display shopping cart contents --%>
26 <%
27    HashMap cart = ( HashMap ) session.getAttribute( "cart" );
28    double total = 0;
29
```

Fig. 11.12 JSP **viewCart.jsp** obtains the shopping cart and outputs an XHTML document with the cart contents in tabular format.

advjhtp1_11.fm Page 554 Saturday, June 9, 2001 11:04 AM


```
30      if ( cart == null || cart.size() == 0 )
31         out.println( "Your shopping cart is currently empty." );
32      else {
33
34         // create variables used in display of cart
35         Set cartItems = cart.keySet();
36         Iterator iterator = cartItems.iterator();
37
38         BookBean book;
39         CartItemBean cartItem;
40
41         int quantity;
42         double price, subtotal;
43
44   %> <%-- end scriptlet for literal XHTML output --%>
45
46      <table>
47         <thead>
48            <th>Product</th>
49            <th>Quantity</th>
50            <th>Price</th>
51            <th>Total</th>
52         </thead>
53
54   <% // continue scriptlet
55
56         while ( iterator.hasNext() ) {
57
58            // get book data; calculate subtotal and total
59            cartItem = ( CartItemBean ) cart.get( iterator.next() );
60            book = cartItem.getBook();
61            quantity = cartItem.getQuantity();
62            price = book.getPrice();
63            subtotal = quantity * price;
64            total += subtotal;
65
66   %> <%-- end scriptlet for literal XHTML and   --%>
67      <%-- JSP expressions output from this loop --%>
68
69            <%-- display table row of book title, quantity, --%>
70            <%-- price and subtotal --%>
71            <tr>
72               <td><%= book.getTitle() %></td>
73
74               <td><%= quantity %></td>
75
76               <td class = "right">
77                  <%=
78                     new DecimalFormat( "0.00" ).format( price )
79                  %>
80               </td>
81
82               <td class = "bold right">
```

Fig. 11.12 JSP **viewCart.jsp** obtains the shopping cart and outputs an XHTML document with the cart contents in tabular format.

advjhtp1_11.fm Page 555 Saturday, June 9, 2001 11:04 AM


```
                  <%=
                     new DecimalFormat( "0.00" ).format( subtotal )
                  %>
               </td>
            </tr>

<% // continue scriptlet

         }  // end of while loop
      }  // end of else

      // make current total a session attribute
      session.setAttribute( "total", new Double( total ) );

%> <%-- end scriptlet --%>

         <%-- display table row containing shopping cart total --%>
         <tr>
            <td colspan = "4" class = "bold right">Total:
               <%= new DecimalFormat( "0.00" ).format( total ) %>
            </td>
         </tr>
      </table>

      <!-- link back to books.jsp to continue shopping -->
      <p class = "bold green">
         <a href = "books.jsp">Continue Shopping</a>
      </p>

      <!-- form to proceed to checkout -->
      <form method = "get" action = "order.html">
         <input type = "submit" value = "Check Out" />
      </form>
</body>

</html>
```

Fig. 11.12 JSP **viewCart.jsp** obtains the shopping cart and outputs an XHTML document with the cart contents in tabular format.

advjhtp1_11.fm Page 556 Saturday, June 9, 2001 11:04 AM

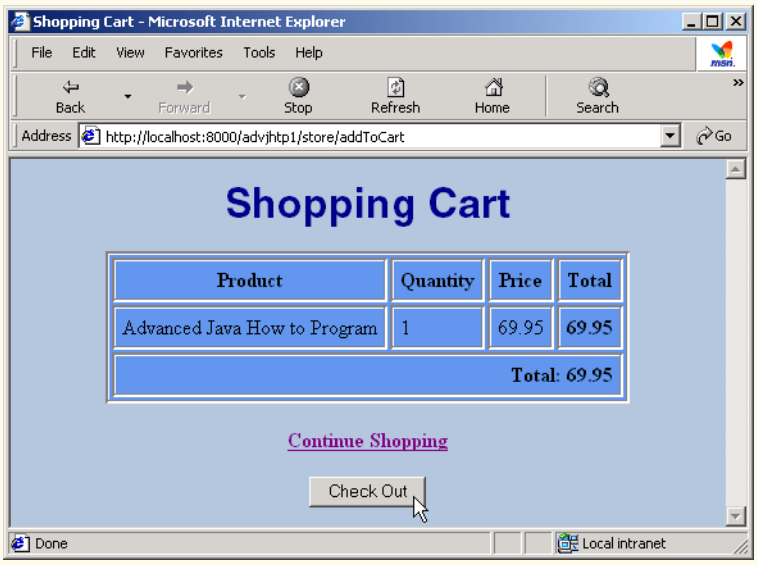

Fig. 11.12 JSP **`viewCart.jsp`** obtains the shopping cart and outputs an XHTML document with the cart contents in tabular format.

The scriptlet at lines 26–44 begins by retrieving the session attribute for the shopping cart **`HashMap`** (line 27). If there is no shopping cart, the JSP simply outputs a message indicating that the cart is empty. Otherwise, lines 35–42 create the variables used to obtain the information that is displayed in the resulting XHTML document. In particular, line 35 obtains the **`Set`** of keys in **`HashMap cart`**. These are used to retrieve the **`CartItemBean`** that represents each book in the cart.

Lines 46–52 output the literal XHTML markup that begins the table that appears in the document. Lines 56–64 continue the scriptlet with a loop that uses each key in the **`HashMap`** to obtain the corresponding **`CartItemBean`**, extracts the data from that bean, calculates the dollar subtotal for that product and calculates the dollar total of all products so far. The last part of the loop body appears outside the scriptlet at lines 71–87 in which the preceding data is formatted into a row in the XHTML table. JSP expressions are used to place each data value in the appropriate table cell. After the loop completes (line 90), line 95 sets a session attribute containing the total of all items in the cart. This value is used by **`process.jsp`** (Fig. 11.14) to display the total as part of the order-processing confirmation. Line 102 outputs the total of all items in the cart as the last row in the XHTML table.

From the XHTML document produced in this JSP, the user can either continue shopping or click the **Check Out** button to proceed to the **`order.html`** ordering page (Fig. 11.13).

11.8 Checking Out

When viewing the cart, the user can click a **Check Out** button to view **`order.html`** (Fig. 11.13). In this example, the form has no functionality. However, it is provided to help

advjhtp1_11.fm Page 557 Saturday, June 9, 2001 11:04 AM

complete the application. Normally, there would be some client-side validation of the form elements, some server-side validation of form elements or a combination of both. When the user presses the button, the browser requests **`process.jsp`** to finalize the book order.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "DTD/xhtml1-strict.dtd">
<!-- order.html -->

<html xmlns = "http://www.w3.org/1999/xhtml"
   xml:lang = "en" lang = "en">

<head>
   <title>Order</title>

   <link rel = "stylesheet" href = "styles.css"
      type = "text/css" />
</head>

<body>
   <p class = "bigFont">Shopping Cart Check Out</p>

   <!-- Form to input user information and credit card.    -->
   <!-- Note: No need to input real data in this example. -->
   <form method = "post" action = "process.jsp">

      <p style = "font-weight: bold">
         Please input the following information.</p>

      <!-- table of form elements -->
      <table>
         <tr>
            <td class = "right bold">First name:</td>

            <td>
               <input type = "text" name = "firstname"
                  size = "25" />
            </td>
         </tr>

         <tr>
            <td class = "right bold">Last name:</td>

            <td>
               <input type = "text" name = "lastname"
                  size = "25" />
            </td>
         </tr>

         <tr>
```

Fig. 11.13 Order form in which the user inputs name, address and credit-card information to complete an order.

advjhtp1_11.fm Page 558 Saturday, June 9, 2001 11:04 AM


```
47              <td class = "right bold">Street:</td>
48
49              <td>
50                 <input type = "text" name = "street" size = "25" />
51              </td>
52           </tr>
53
54           <tr>
55              <td class = "right bold">City:</td>
56
57              <td>
58                 <input type = "text" name = "city" size = "25" />
59              </td>
60           </tr>
61
62           <tr>
63              <td class = "right bold">State:</td>
64
65              <td>
66                 <input type = "text" name = "state" size = "2" />
67              </td>
68           </tr>
69
70           <tr>
71              <td class = "right bold">Zip code:</td>
72
73              <td>
74                 <input type = "text" name = "zipcode"
75                    size = "10" />
76              </td>
77           </tr>
78
79           <tr>
80              <td class = "right bold">Phone #:</td>
81
82              <td>
83                 (
84                    <input type = "text" name = "phone" size = "3" />
85                 )
86
87                 <input type = "text" name = "phone2"
88                        size = "3" /> -
89
90                 <input type = "text" name = "phone3" size = "4" />
91              </td>
92           </tr>
93
94           <tr>
95              <td class = "right bold">Credit Card #:</td>
96
97              <td>
98                 <input type = "text" name = "creditcard"
99                    size = "25" />
```

Fig. 11.13 Order form in which the user inputs name, address and credit-card information to complete an order.

advjhtp1_11.fm Page 559 Saturday, June 9, 2001 11:04 AM


```
100               </td>
101            </tr>
102
103            <tr>
104               <td class = "right bold">Expiration (mm/yy):</td>
105
106               <td>
107                  <input type = "text" name = "expires"
108                     size = "2" /> /
109
110                  <input type = "text" name = "expires2"
111                     size = "2" />
112               </td>
113            </tr>
114         </table>
115
116         <!-- enable user to submit the form  -->
117         <p><input type = "submit" value = "Submit" /></p>
118      </form>
119   </body>
120
121   </html>
```

Fig. 11.13 Order form in which the user inputs name, address and credit-card information to complete an order.

advjhtp1_11.fm Page 560 Saturday, June 9, 2001 11:04 AM

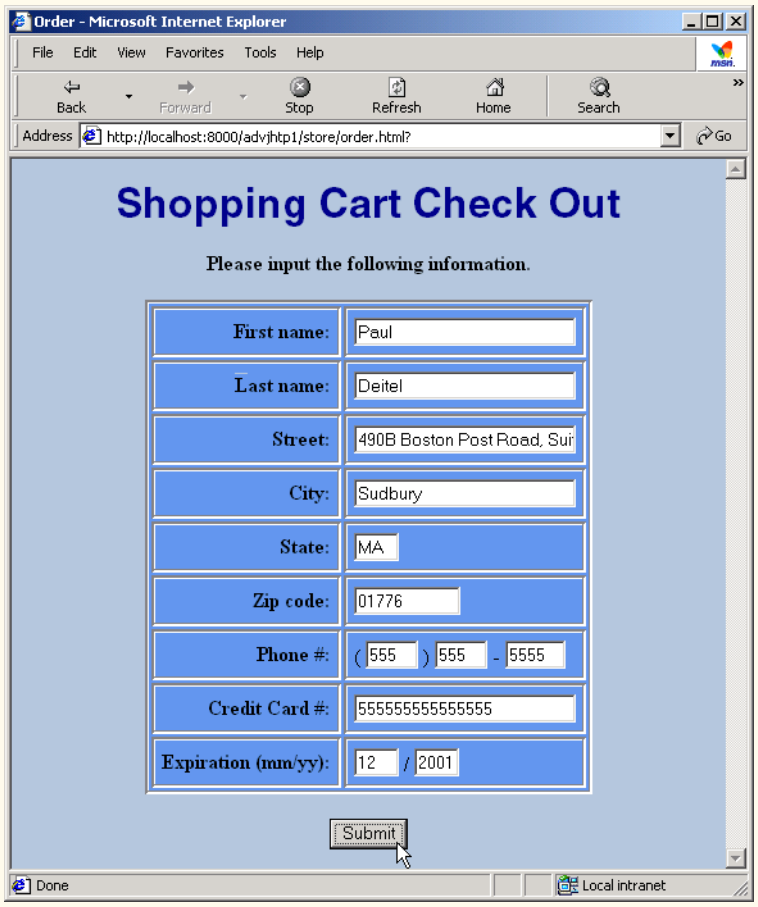

Fig. 11.13 Order form in which the user inputs name, address and credit-card information to complete an order.

11.9 Processing the Order

JSP **`process.jsp`** (Fig. 11.14) pretends to process the user's credit-card information and creates an XHTML document containing a message that the order was processed and the final order dollar total. The scriptlet at lines 20–29 obtains the session attribute **`total`**. The **`Double`** object returned is converted to a **`double`** and stored in Java variable **`total`**. Our simulation of a bookstore does not perform real credit-card processing, so the transaction is now complete. Therefore, line 27 invokes **`HttpSession`** method **`invalidate`** to discard the session object for the current client. In a real store, the session would not be invalidated until the purchase is confirmed by the credit-card company. Lines 31–41 define the body of the XHTML document sent to the client. Line 38 uses a JSP expression to insert the total of all items purchased.

advjhtp1_11.fm Page 561 Saturday, June 9, 2001 11:04 AM


```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "DTD/xhtml1-strict.dtd">
<!-- process.jsp -->

<%-- JSP page settings --%>
<%@ page language = "java" session = "true" %>
<%@ page import = "java.text.*" %>

<html xmlns = "http://www.w3.org/1999/xhtml"
   xml:lang = "en" lang = "en">

<head>
   <title>Thank You!</title>

   <link rel = "stylesheet" href = "styles.css"
      type = "text/css" />
</head>

<% // start scriptlet

   // get total order amount
   Double d = ( Double ) session.getAttribute( "total" );
   double total = d.doubleValue();

   // invalidate session because processing is complete
   session.invalidate();

%> <%-- end scriptlet --%>

<body>
   <p class = "bigFont">Thank You</p>

   <p>Your order has been processed.</p>

   <p>Your credit card has been billed:
      <span class = "bold">
         $<%= new DecimalFormat( "0.00" ).format( total ) %>
      </span>
   </p>
</body>

</html>
```

Fig. 11.14 JSP **process.jsp** performs the final order processing.

advjhtp1_11.fm Page 562 Saturday, June 9, 2001 11:04 AM

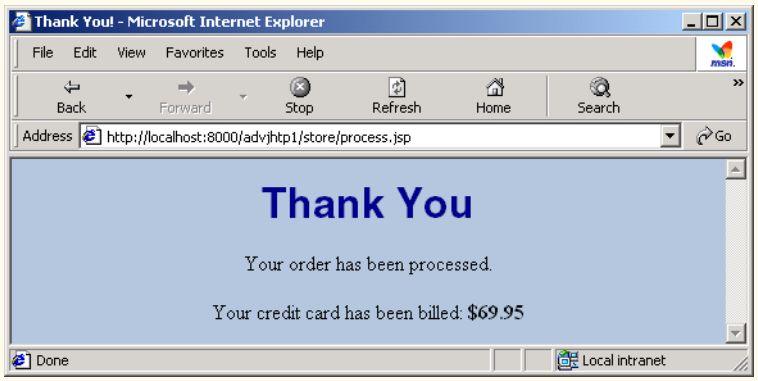

Fig. 11.14 JSP **process.jsp** performs the final order processing.

11.10 Deploying the Bookstore Application in J2EE 1.2.1

Next, we deploy the bookstore application in the Java 2 Enterprise Edition 1.2.1 reference implementation. This section assumes that you have downloaded and installed J2EE 1.2.1. If not, please refer to Appendix G now for installation details. Note that the files for this entire bookstore application can be found on the CD that accompanies this book and at **www.deitel.com**.

In Sections 13.10.1 through 13.10.8 we demonstrate the steps needed to perform to deploy this application:

1. Configure the **books** data source for use with the J2EE 1.2.1 reference implementation server.
2. Launch the Cloudscape database server and the J2EE 1.2.1 reference implementation server for deployment and execution of the application.
3. Launch the **Application Deployment Tool**. This tool provides a graphical user interface for deploying applications on the J2EE 1.2.1 server.
4. Create a new application in the **Application Deployment Tool**.
5. Add library JAR files to the application. These are available to all application components.
6. Create a new Web component in the application for the **BookServlet**.
7. Create a new Web component in the application for the **AddToCartServlet**.
8. Add the non-servlet components to the application. These include XHTML documents, JSPs, images, CSS files, XSL files and JavaBeans used in the application.
9. Specify the Web context that causes this J2EE application to execute. This determines the URL that will be used to invoke the application.
10. Specify the database resource (i.e., **books**) used by our application.

advjhtp1_11.fm Page 563 Saturday, June 9, 2001 11:04 AM

11. Set up the JNDI name for the database in the application. This is used to register the name with the Java Naming and Directory Service so the database can be located at execution time.
12. Set up the *welcome file* for our application. This is the initial file that is returned when the user invokes the bookstore application.
13. Deploy the application.
14. Run the application.

At the end of this Section 11.10.8, you will be able to deploy and test the bookstore application.

11.10.1 Configuring the `books` Data Source

Before deploying the bookstore application, you must configure the **`books`** data source so the J2EE server registers the data source with the naming server. This enables the application to use JNDI to locate the data source at execution time. J2EE comes with Cloudscape—a pure-Java database application from Informix Software. We use Cloudscape to perform our database manipulations in this case study.

To configure the Cloudscape data source, you must modify the J2EE default configuration file **`default.properties`** in the J2EE installation's **`config`** directory. Below the comment **`JDBC URL Examples`** is a line that begins with **`jdbc.datasources`**. Append the following text to this line

```
|jdbc/books|jdbc:cloudscape:rmi:books;create=true
```

The vertical bar, **`|`**, at the beginning of this text separates the new data source we are registering from a data source that is registered by default when you install J2EE. The text **`jdbc/books`** is the JNDI name for the database. After the second **`|`** character in the preceding text is the JDBC URL **`jdbc:cloudscape:rmi:books`**. The URL indicates that the J2EE will use the JDBC protocol to interact with the Cloudscape subprotocol, which, in turn, uses RMI to communicate with the database (**`books`** in this case). Finally, **`create=true`** specifies that J2EE should create a database if the database does not already exist. [Remember, that the **`books`** database was created in Chapter 8.] After configuring the database, save the **`default.properties`** file. This completes *Step 1* of Section 11.10.

Portability Tip 11.2

Each database server typically has its own URL format that enables an application to interact with databases hosted on that database server. See your database server's documentation for more information.

11.10.2 Launching the Cloudscape Database and J2EE Servers

Step 2 of Section 11.10 specifies that you must launch the Cloudscape database server and the J2EE server, so you can deploy and execute the application. Open a command prompt (or shell) and change directories to the **`bin`** subdirectory of your J2EE installation. Then, issue the following commands:

advjhtp1_11.fm Page 564 Saturday, June 9, 2001 11:04 AM


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

The first command starts the Cloudscape database server. The second command starts the J2EE server. Note that the J2EE server includes the Tomcat JSP and servlet container discussed in Chapter 9.

Portability Tip 11.3

On some UNIX/Linux systems, you may need to precede the commands that launch the Cloudscape server and the J2EE server with `./` *to indicate that the command is located in the current directory.*

Testing and Debugging Tip 11.1

Use separate command prompts (or shells) to execute the commands that launch the Cloudscape database server and the J2EE 1.2.1 server, so you can see any error messages generated by these programs.

Testing and Debugging Tip 11.2

To ensure that the J2EE server communicates properly with the Cloudscape server (or any other database server), always launch the database server before the J2EE server. Otherwise, exceptions will occur when the J2EE server attempts to configure its data sources.

To shutdown the servers, use a command prompt (or shell) to execute the following commands:

```
j2ee -stop
cloudscape -stop
```

Testing and Debugging Tip 11.3

Always shut down the J2EE server before the Cloudscape database server to ensure that the J2EE server does not attempt to communicate with the Cloudscape database server after the database server has been shut down. If Cloudscape is terminated first, it is possible that the J2EE server will receive another request at attempt to access the database again. This will result in exceptions.

11.10.3 Launching the J2EE Application Deployment Tool

Step 3 of Section 11.10 begins the process of deploying our bookstore application. The J2EE reference implementation comes with a graphical application called the **Application Deployment Tool** that helps you deploy Enterprise Java applications. In Chapter 9, we created an XML deployment descriptor by hand to deploy our servlets. The **Application Deployment Tool** is nice in that it writes the deployment descriptor files for you and automatically archives the Web application's components. The tool places all Web application components and auxiliary files for a particular application into a single *Enterprise Application Archive (EAR) file*. This file contains deployment descriptor information, WAR files with Web application components and additional information that is discussed later in the book.

Execute the deployment tool by opening a command prompt (or shell) and changing directories to the `bin` subdirectory of your J2EE installation. Then, type the following command:

advjhtp1_11.fm Page 565 Saturday, June 9, 2001 11:04 AM


```
deploytool
```

The **Application Deployment Tool** window (Fig. 11.15) appears. [*Note:* In our deployment discussion, we cover only those aspects of the deployment tool required to deploy this bookstore application. Later in the book we discuss other aspects of this tool in detail.]

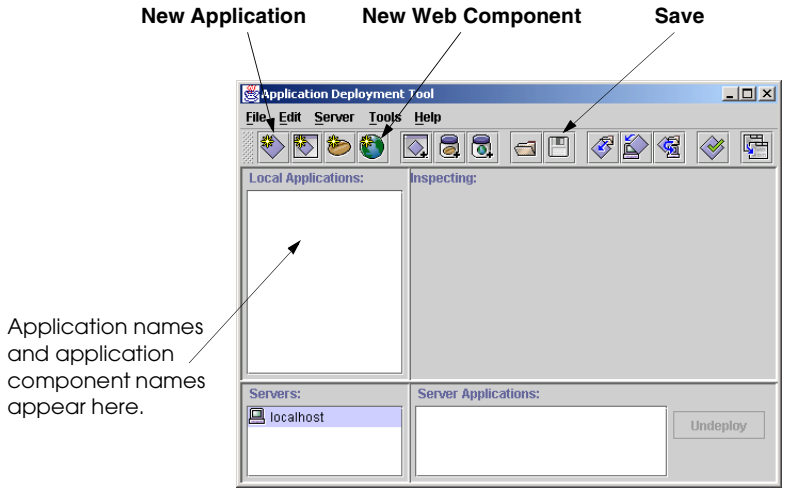

Fig. 11.15 **Application Deployment Tool** main window.

11.10.4 Creating the Bookstore Application and Adding Library JARs

The **Application Deployment Tool** simplifies the task of deploying Enterprise applications. Next (*Step 4* of Section 11.10) create the new application. Click the ***New Application*** *button* to display the ***New Application*** *window* (Fig. 11.16).

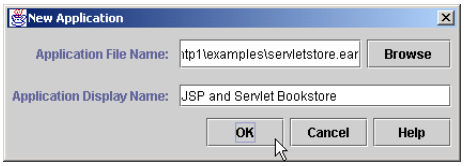

Fig. 11.16 **New Application** window.

In the ***Application File Name*** *field* you can type the name of the EAR file in which the **Application Deployment Tool** stores the application components, or you can click **Browse** to specify both the name and location of the file. In the ***Application Display Name*** *field*, you can specify the name for your application. This will appear in the **Local Applications** area of the deployment tool's main window (Fig. 11.15). Click **OK** to create the application. The main **Application Deployment Tool** window now appears as shown in Fig. 11.17.

advjhtp1_11.fm Page 566 Saturday, June 9, 2001 11:04 AM

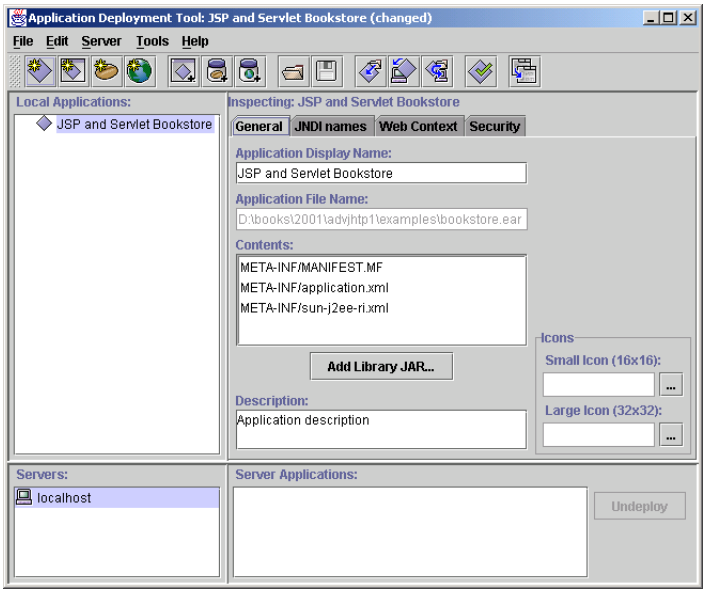

Fig. 11.17 **Application Deployment Tool** main window after creating a new application.

Our bookstore application's **`BookServlet`** and **`BookBean`** classes use XML APIs from Sun's Java API for XML Parsing (JAXP) 1.1 that are not currently part of the Java 2 Standard Edition or the Java 2 Enterprise Edition. For this reason, we must include three JAR files **`crimson.jar`**, **`jaxp.jar`** and **`xalan.jar`** from JAXP 1.1. These are located in the root directory of your JAXP 1.1 installation (normally called **`jaxp-1.1`**). Remember that the three JARs also must be copied into the Java extensions mechanism directory for your Java 2 Standard Edition installation (**`jre/lib/ext`** on UNIX/Linux or **`jre\lib\ext`** on Windows).

To add the JARs to our bookstore application (*Step 5* of Section 11.10), click the ***Add Library JAR...*** *button* in the middle of the **Application Deployment Tool** window (Fig. 11.18). Locate your **`jaxp-1.1`** directory in the file dialog, select **`crimson.jar`** and click the ***Add Library JAR*** *button* in the bottom-right corner of the dialog. Repeat these steps for the **`jaxp.jar`** and **`xalan.jar`** files. After performing this step, the **Application Deployment Tool** window should appear as in Fig. 11.20. In the **Contents** area of the window, note that the **Application Deployment Tool** placed the library JARS in a directory called **`library`**.

advjhtp1_11.fm Page 567 Saturday, June 9, 2001 11:04 AM

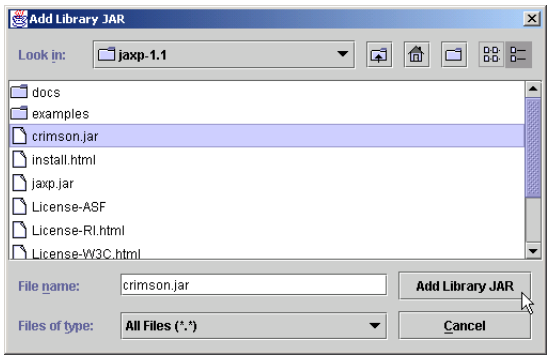

Fig. 11.18 **Add Library JAR** window.

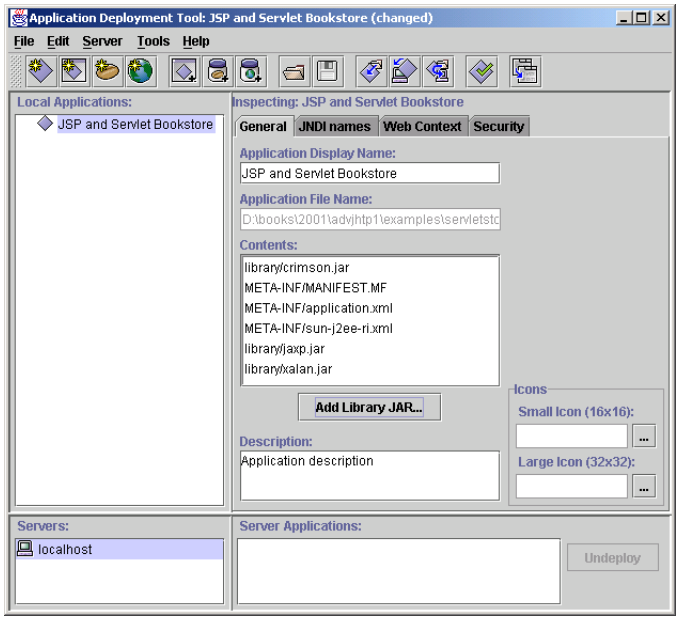

Fig. 11.19 **Application Deployment Tool** main window after adding library JARs.

11.10.5 Creating `BookServlet` and `AddToCartServlet` Web Components

Step 6 of Section 11.10 is to create Web components for the **`BookServlet`** and the **`AddToCartServlet`**. This will enable us to specify the alias that is used to invoke each servlet. We will show the details of creating the **`BookServlet`** Web component. Then, you can repeat the steps to create the **`AddToCartServlet`** Web component.

advjhtp1_11.fm Page 568 Saturday, June 9, 2001 11:04 AM

To begin, click the ***New Web Component*** *button* (see Fig. 11.15) to display the ***Introduction*** *window* of the ***New Web Component Wizard*** (Fig. 11.20).

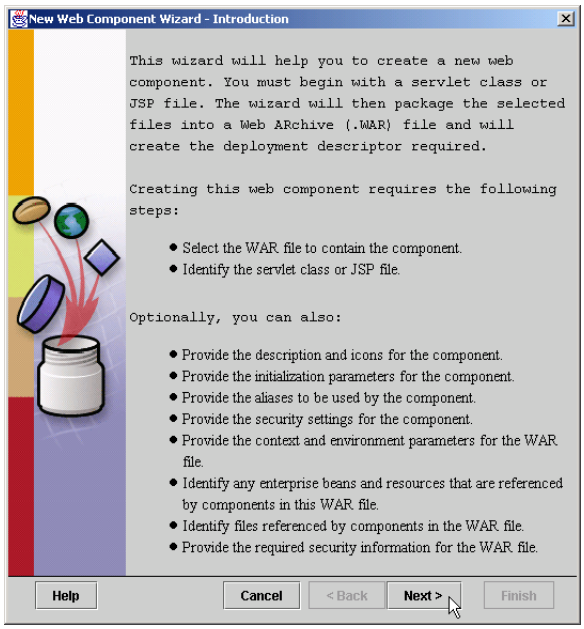

Fig. 11.20 **New Web Component Wizard - Introduction** window.

Click the ***Next >*** *button* to display the ***WAR File General Properties*** (Fig. 11.21) window of the ***New Web Component Wizard***.

advjhtp1_11.fm Page 569 Saturday, June 9, 2001 11:04 AM

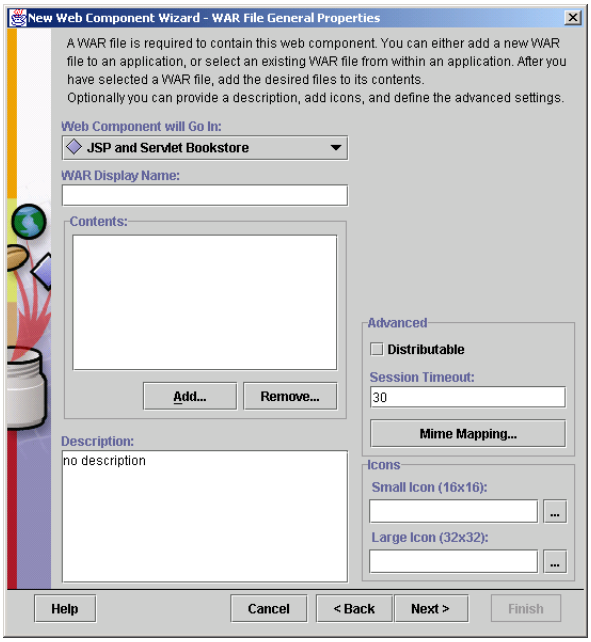

Fig. 11.21 New Web Component Wizard - WAR File General Properties window.

Ensure that **JSP and Servlet Bookstore** is selected in the ***Web Component will Go In:*** *drop-down list*. In the ***WAR Display Name*** *field*, type a name (`Store Components`) for the WAR that will appear in the **Local Applications** area of the deployment tool's main window (see Fig. 11.15). Then, click the ***Add...*** *button* to display the ***Add Files to .WAR - Add Content Files*** *window* (Fig. 11.22). Content files are non-servlet files such as images, XHTML documents, style sheets and JSPs. We will be adding these in another step later, so click **Next >** to proceed to the ***Add Files to .WAR - Add Class Files*** *window* (Fig. 11.27)

advjhtp1_11.fm Page 570 Saturday, June 9, 2001 11:04 AM

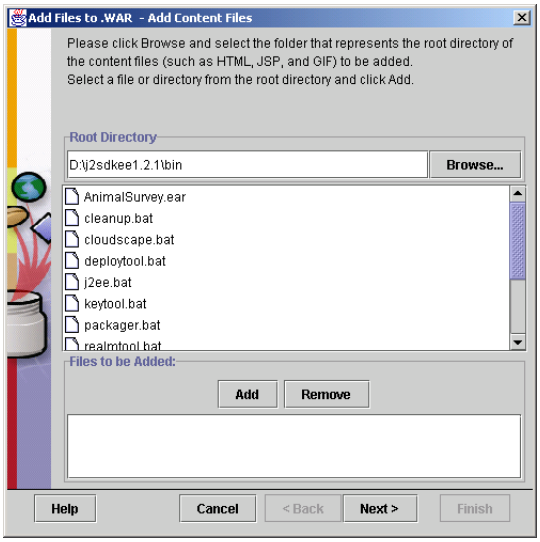

Fig. 11.22 Add Files to .WAR - Add Content Files window.

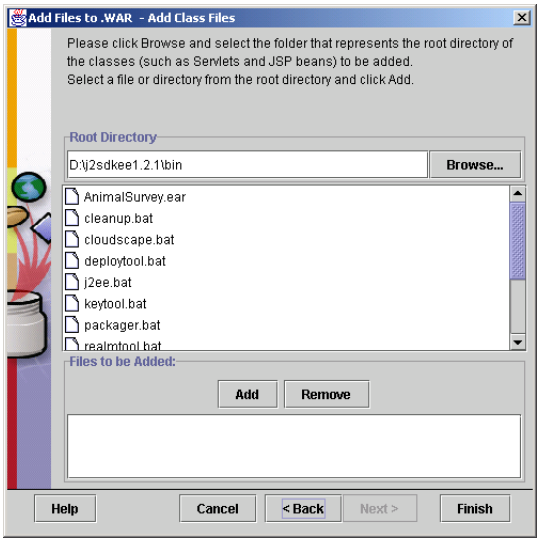

Fig. 11.23 Add Files to .WAR - Add Class Files window.

To add the **`BookServlet.class`** file, click the ***Browse...*** *button* to display the ***Choose Root Directory*** *window* (Fig. 11.24).

advjhtp1_11.fm Page 571 Saturday, June 9, 2001 11:04 AM

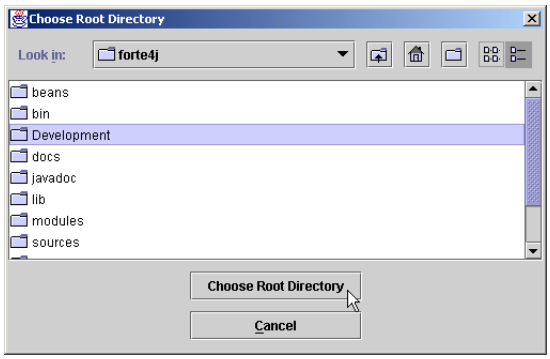

Fig. 11.24 **Choose Root Directory** window.

When adding a class file for a class in a package (as all classes are in this example), it is imperative that the files be added and maintained in their full package directory structure or as part of a JAR file that contains the full package directory structure. In this example, we did not create a JAR file containing the entire **`com.deitel.advjhtp1.store`** package. Therefore, we need to locate the directory in which the first package directory name is found.

On our system, the **`com`** directory that starts the package name is located in a directory called **`Development`**. When you click the ***Choose Root Directory*** *button*, you are returned to the **Add Files to .WAR - Add Class Files** window. In that window, you should be able to locate the **`com`** directory (Fig. 11.25).

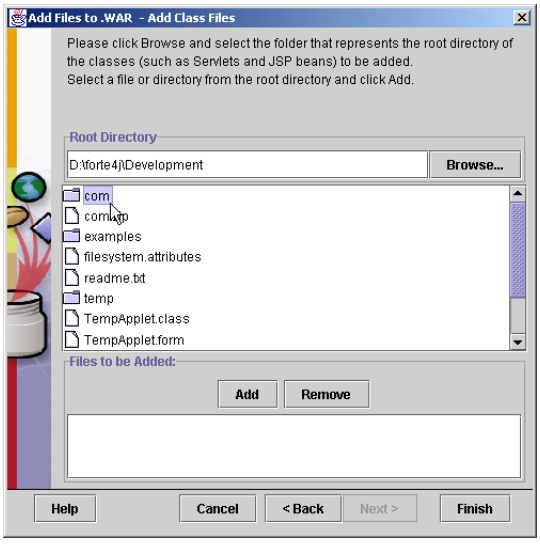

Fig. 11.25 **Add Files to .WAR - Add Class Files** window after selecting the root directory in which the files are located.

advjhtp1_11.fm Page 572 Saturday, June 9, 2001 11:04 AM

Double click the **com** directory name to expand its contents in the window. Do the same for the subdirectory **deitel**, then the subdirectory **advjhtp1** and finally for the directory **store**. In the store directory, select the **.class** file for **BookServlet**, then click the **Add** button. At the bottom of the **Add Files to .WAR - Add Class Files** window, the **.class** file should be displayed with its full package directory structure. After doing this, click the **Finish** button to return the **New Web Component Wizard - WAR File General Properties** window. Note that the file selected with the **Add Files to .WAR** window now appears in the **Contents** text area of the window (Fig. 11.26).

Common Programming Error 11.1

Not including the full package directory structure for a class in a package will prevent the application from loading the class and prevent the application from executing properly.

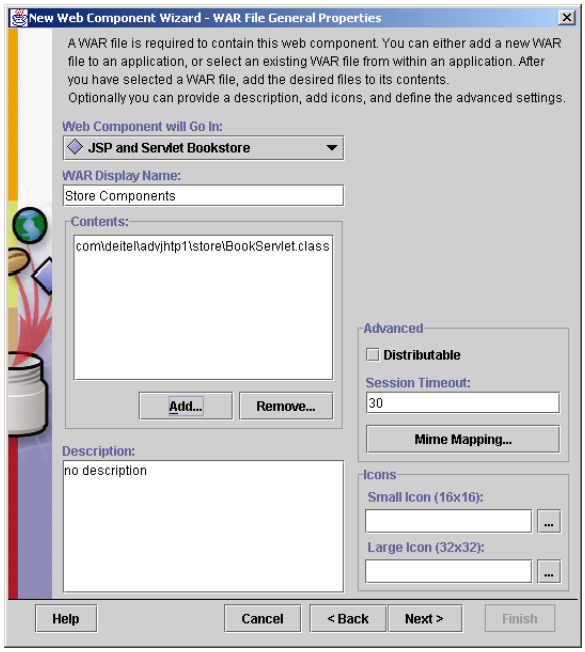

Fig. 11.26 **New Web Component Wizard - WAR File General Properties** window after selecting the file **BookServlet.class**.

Click **Next >** to proceed to the ***New Web Component Wizard - Choose Component Type*** *window* and select ***Servlet*** (Fig. 11.27).

advjhtp1_11.fm Page 573 Saturday, June 9, 2001 11:04 AM

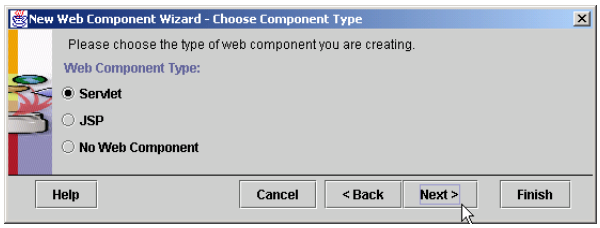

Fig. 11.27 **New Web Component Wizard - Choose Component Type** window.

Click **Next >** to proceed to the ***New Web Component Wizard - Component General Properties*** *window* (Fig. 11.28). Select the **`BookServlet`** class in the ***Servlet Class*** *drop-down list* and type **`Book Servlet`** in the ***Web Component Display Name*** *field*.

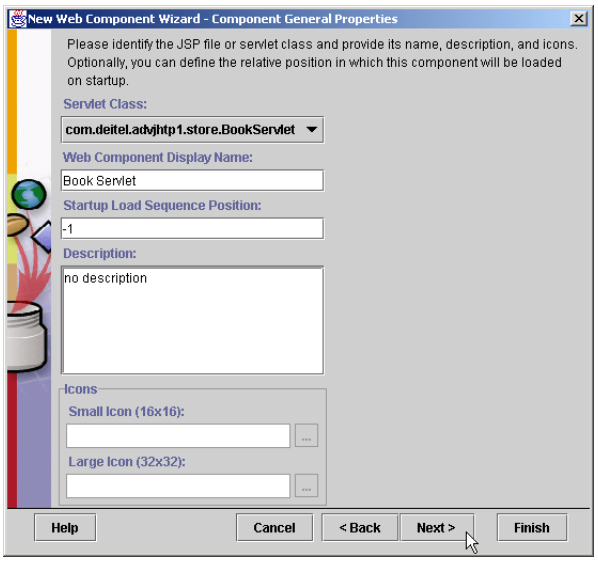

Fig. 11.28 **New Web Component Wizard - Component General Properties** window.

Click **Next >** twice to display the ***New Web Component Wizard - Component Aliases*** *window* (Fig. 11.29). Click **Add** to specify an alias for the **`BookServlet`**. Click the white box that appears in the window, type **`displayBook`** as the alias for the servlet and press *Enter*. Next, press ***Finish*** to complete the setup of the **`BookServlet`**.

advjhtp1_11.fm Page 574 Saturday, June 9, 2001 11:04 AM

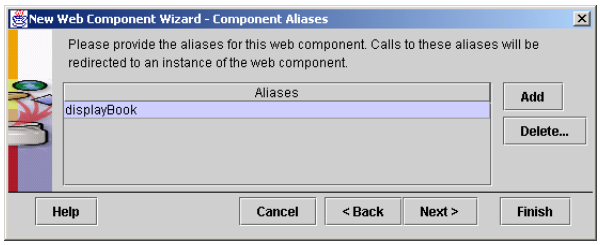

Fig. 11.29 **New Web Component Wizard - Component Aliases** window.

Now, create a Web component for **`AddToCartServlet`** (*Step 7* of Section 11.10) by repeating the steps shown in this section. For this Web component, specify **`Add to Cart Servlet`** as the **Web Component Display Name** and **`addToCart`** as the alias for the servlet. After adding the two servlet Web components, the **Application Deployment Tool** window should appear as shown in Fig. 11.30.

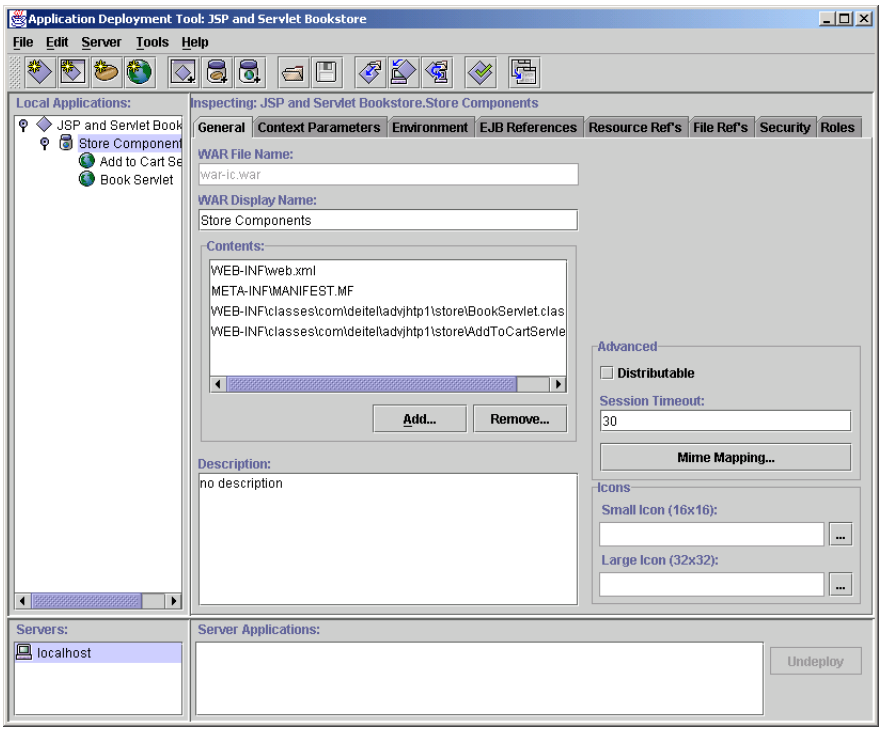

Fig. 11.30 **Application Deployment Tool** window after deploying **`BookServlet`** and **`AddToCartServlet`**.

advjhtp1_11.fm Page 575 Saturday, June 9, 2001 11:04 AM

11.10.6 Adding Non-Servlet Components to the Application

Next, we will add our non-servlet components to the application (*Step 8* of Section 11.10). These files include JSPs, XHTML documents, style sheets, images and JavaBeans used in the application.

Begin by expanding the application component tree and clicking **Store Components** in the ***Local Applications*** area of the **Application Deployment Tool** window (see Fig. 11.30). In the contents area of the **Application Deployment Tool** window, click the **Add...** button to display the **Add Files to .WAR - Add Content Files** window (Fig. 11.31).

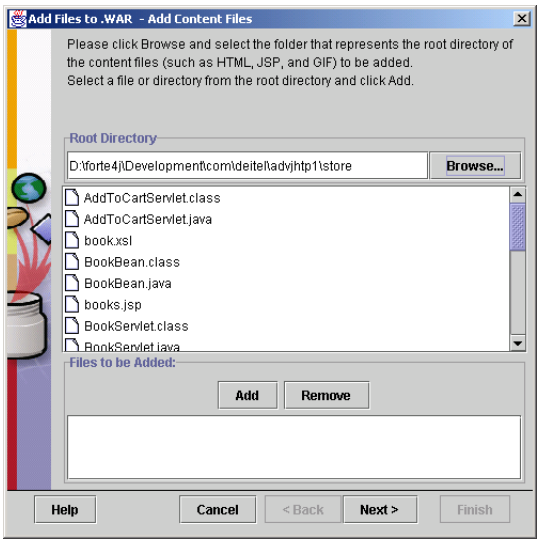

Fig. 11.31 **Add Files to .WAR - Add Content Files** window.

Navigate to the directory on your system that contains the files for the bookstore application. In the list box that appears in the window, locate each of the following files and directories: **`book.xsl`**, **`books.jsp`**, **`images`**, **`index.html`**, **`order.html`**, **`process.jsp`**, **`styles.css`** and **`viewCart.jsp`**. For each file or directory, click the **Add** button. You can select multiple items at one time by holding the <*Ctrl*> key and clicking each item. All the items you add should appear in the text area at the bottom of this window. When you are done, click **Next >** to display the **Add Files to .WAR - Add Class Files** window (Fig. 11.32).

advjhtp1_11.fm Page 576 Saturday, June 9, 2001 11:04 AM

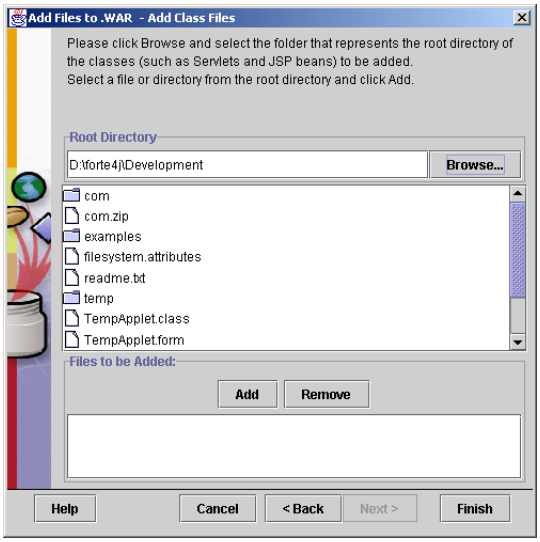

Fig. 11.32 **Add Files to .WAR - Add Class Files** window.

We will use this window to add the **`.class`** files for the non-servlet classes (i.e., our JavaBeans) to our application. Remember that the JavaBeans used in the bookstore are in a package, so their **`.class`** files must be added and maintained in their full package directory structure. Once again, click the **Browse...** button to display the **Choose Root Directory** window and locate the directory in which the first package directory name is found. Select that directory as the root directory. Double click the **`com`** directory name to expand its contents in the window. Do the same for the subdirectory **`deitel`**, then the subdirectory **`advjhtp1`** and finally for the directory **`store`**. In the store directory, select the **`.class`** files for each of the JavaBeans in this bookstore example (**`BookBean.class`**, **`CartItemBean.class`** and **`TitlesBean.class`**), then click the **Add** button. At the bottom of the **Add Files to .WAR - Add Class Files** window, each **`.class`** file should be displayed with its full package directory structure. After doing this, click the **Finish** button to return the **Application Deployment Tool** window. Note that the files selected with the **Add Files to .WAR** windows now appear in the **Contents** text area of the window. Click the **Save** button to save your work.

11.10.7 Specifying the Web Context, Resource References, JNDI Names and Welcome Files

Steps 9 through 13 of Section 11.10 perform the final configuration and deployment of the bookstore application. After performing the steps in this section, you will be able to execute the bookstore application.

We begin by specifying the Web context for our application (*Step 9* of Section 11.10). At the beginning of this chapter, we indicated that the user would enter the URL

```
http://localhost:8000/advjhtp1/store
```

advjhtp1_11.fm Page 577 Saturday, June 9, 2001 11:04 AM

in a browser to access the bookstore application. The Web context is the part of the preceding URL that enables the server to determine which application to execute when the server receives a request from a client. In this case, the Web context is **`advjhtp1/store`**. Once again, note that the J2EE server uses port 8000, rather than the port 8080 used by Tomcat.

Common Programming Error 11.2

Specifying the wrong port number in a URL that is supposed to access the J2EE server causes your Web browser to indicate that the server was not found.

Testing and Debugging Tip 11.4

When deploying an Enterprise Java application on a production application server (the J2EE server is for testing only), it is typically not necessary to specify a port number in the URL when accessing the application. See your application server's documentation for further details.

To specify the Web context, click the **JSP and Servlet Bookstore** node in the **Local Applications** area of the **Application Deployment Tool** window. Then, click the ***Web Context*** *tab* (Fig. 11.33). Click the white box in the ***Context Root*** *column* and type **`advjhtp1/store`**, then press *Enter*.

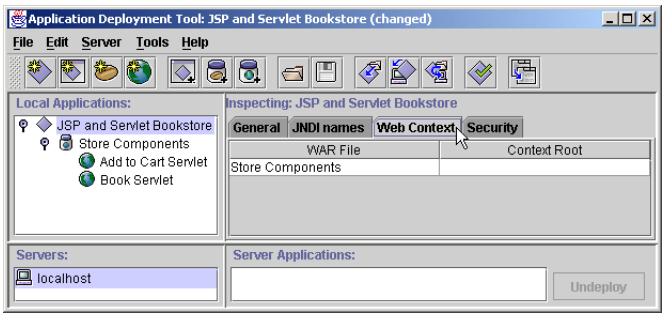

Fig. 11.33 Specifying the **Web Context** in the **Application Deployment Tool**.

Next, we must specify the database resource referenced by the bookstore application (*Step 10* of Section 11.10). Click the **Store Components** node in the **Local Applications** area of the **Application Deployment Tool** window. Then, click the ***Resource Ref's*** *tab* (Fig. 11.34). Click the **Add** button. Under the ***Coded Name*** *column* click the white box and type **`jdbc/books`** (the JNDI name of our data source). Figure 11.34 shows the **Application Deployment Tool** window after creating the resource reference.

advjhtp1_11.fm Page 578 Saturday, June 9, 2001 11:04 AM

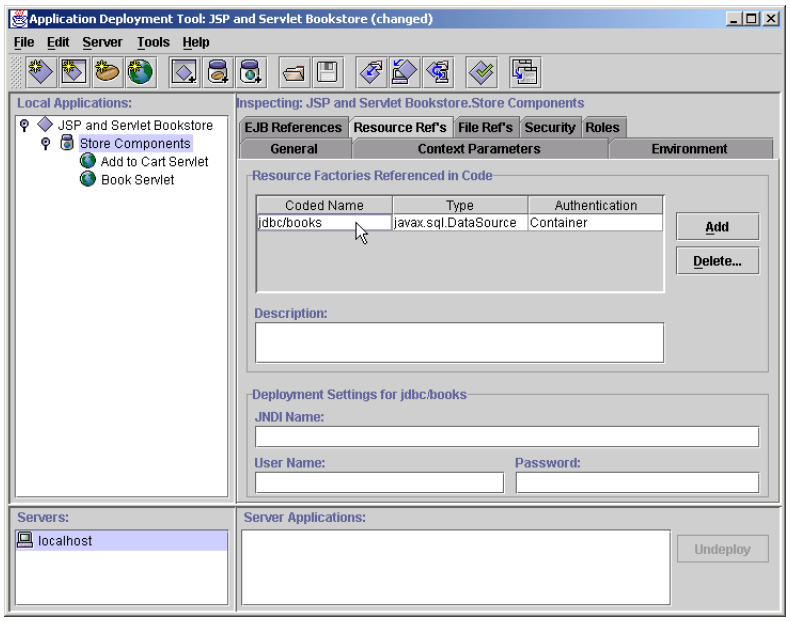

Fig. 11.34 Specifying the **Resource Ref's** in the **Application Deployment Tool**.

Next, we specify the JNDI name for the database in the application (*Step 11* of Section 11.10). This is used to register the name with the Java Naming and Directory Service, so the database can be located by the application at execution time. To specify the JNDI name for the database, click the **JSP and Servlet Bookstore** node in the **Local Applications** area of the **Application Deployment Tool** window. Then, click the ***JNDI names*** *tab* (Fig. 11.35). In the ***JNDI Name*** *column*, click the white box and type **`jdbc/books`** then press *Enter*.

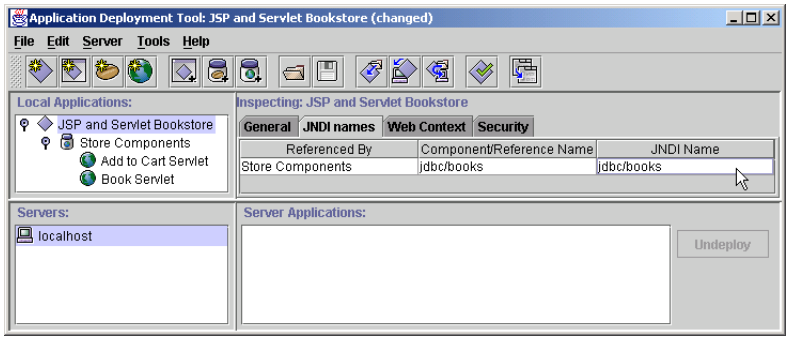

Fig. 11.35 Specifying the **Resource Ref's** in the **Application Deployment Tool**.

advjhtp1_11.fm Page 579 Saturday, June 9, 2001 11:04 AM

The last task to perform before deploying the application is specifying the welcome file that is displayed when the user first visits the bookstore. Click the **Store Components** node in the **Local Applications** area of the **Application Deployment Tool** window. Then, click the ***File Ref's*** *tab* (Fig. 11.36). Click the **Add** button. In the ***Welcome Files*** area click the white box and type **`index.html`**. Figure 11.36 shows the **Application Deployment Tool** window after specifying the welcome file. Click the **Save** button to save the application settings.

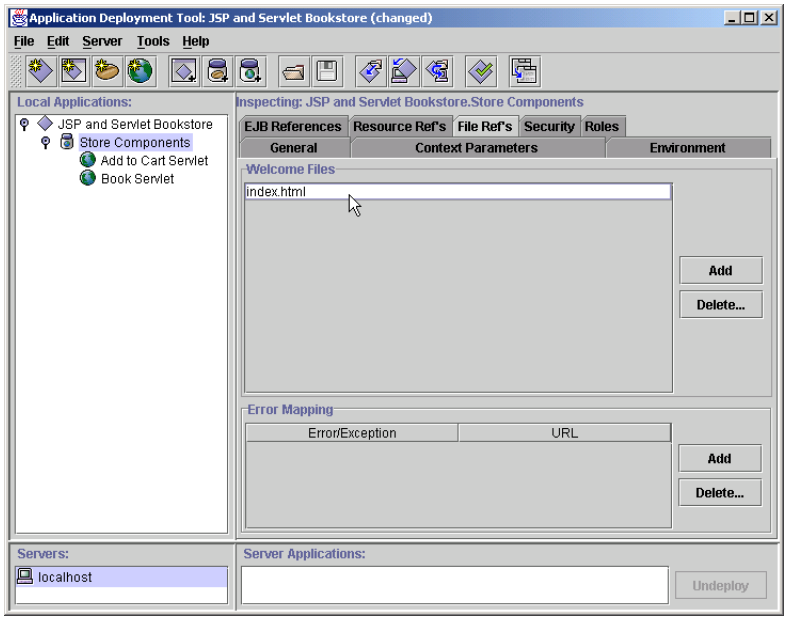

Fig. 11.36 Specifying the welcome file in the **File Ref's** tab of the **Application Deployment Tool**.

11.10.8 Deploying and Executing the Application

Now, you are ready to deploy the bookstore application so you can test it. Figure 11.37 shows **Application Deployment Tool** toolbar buttons for updating application files and deploying applications. The ***Update Application Files*** *button* updates the application's EAR file after changes are made to any of the files, such as recompiling classes or modifying files. The ***Deploy Application*** *button* causes the **Application Deployment Tool** to communicate with the J2EE server and deploy the application. The functionality of both these buttons is combined in the ***Update and Redeploy Application*** *button*.

advjhtp1_11.fm Page 580 Saturday, June 9, 2001 11:04 AM

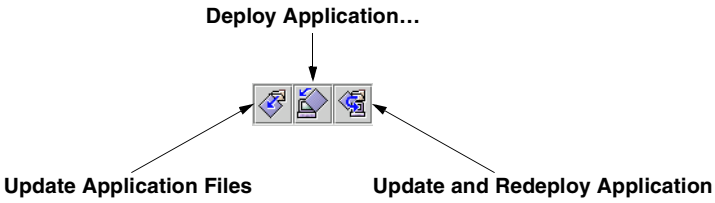

Fig. 11.37 **Application Deployment Tool** toolbar buttons for updating application files and deploying applications.

Click the **Deploy Application** button to display the **Deploy JSP and Servlet Bookstore - Introduction** window (Fig. 11.38). Press the **Next >** button three times, then press the **Finish** button. A ***Deployment Progress*** *window* appears. This window will indicate when the deployment is complete. When that occurs, open a Web browser and enter the following URL to test the bookstore application:

```
http://localhost:8000/advjhtp1/store
```

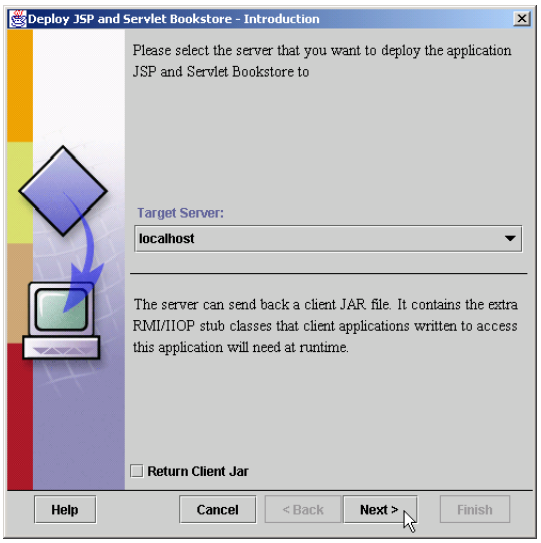

Fig. 11.38 **Deploy JSP and Servlet Bookstore - Introduction** window.

In this chapter, we presented our first substantial Enterprise Java application. The steps presented in this section for deploying the bookstore application are just some of the steps required in a typical Enterprise application. For example, there were no security requirements in the bookstore application. In real Enterprise Java applications, some or all of the application components have security restrictions such as "the user must enter a valid username and password before access is granted to a component." Such restrictions are specified at deployment time with the **Application Deployment Tool** or some similar tool in

advjhtp1_11.fm Page 581 Saturday, June 9, 2001 11:04 AM

an Enterprise Java development environment. These security restrictions are enforced by the application server. In our bookstore example, if the JSPs had security restrictions, it would be necessary to deploy each one individually as we did with the **`BookServlet`** and **`AddToCartServlet`**. In later chapters, we discuss more of the deployment options for application components. The *Java 2 Enterprise Edition Specification* (available at **`java.sun.com/j2ee/download.html`**) discusses the complete set of deployment options that are required in J2EE-compliant application servers. The next chapter, Wireless Internet and M-Business, continues our client/server discussions. In that chapter, we use servlets and XML to create content for wireless devices such as pagers, cell phones and personal digital assistants.

SUMMARY

- The Java 2 Enterprise Edition 1.2.1 reference implementation includes the Apache Tomcat JSP and servlet container.
- A three-tier, distributed Web application consists of client, server and database tiers.
- The client tier in a Web application often is represented by the user's Web browser.
- The server tier in a Web application often consists of JSPs and servlets that act on behalf of the client to perform tasks.
- The database tier maintains the database accessed from the server tier.
- The Java 2 Enterprise Edition 1.2.1 reference implementation comes with an **Application Deployment Tool**. Among its many features, this tool enables us to specify the alias used to invoke a servlet.
- The **Application Deployment Tool** creates the deployment descriptor for a servlet as part of deploying an application.
- The welcome file is the default document sent as the response to a client when the client initially interacts with a J2EE application.
- Different browsers have different levels of support for Cascading Style Sheets.
- JavaServer Pages often generate XHTML that is sent to the client for rendering.
- The Java Naming and Directory Interface (JNDI) enables Enterprise Java application components to access information and resources (such as databases) that are external to an application. In some cases, those resources are distributed across a network.
- An Enterprise Java application container must provide a naming service that implements JNDI and enables the components executing in that container to perform name lookups to locate resources. The J2EE 1.2.1 reference implementation server includes such a naming service.
- When you deploy an Enterprise Java application, you specify the resources used by the application (such as databases) and the JNDI names for those resources.
- Using an **`InitialContext`**, an Enterprise application component can lookup a resource. The **`InitialContext`** provides access to the application's naming environment.
- **`InitialContext`** method **`lookup`** locates a resource with a JNDI name. Method **`lookup`** returns a **`DataSource`** object and throws a **`NamingException`** if it cannot resolve the name it receives as an argument.
- A **`DataSource`** is used to connect to a database.
- The **`org.w3c.dom`** package's **`Document`** and **`Element`** interfaces are used to create an XML document tree.

advjhtp1_11.fm Page 582 Saturday, June 9, 2001 11:04 AM

- **`Document`** method **`createElement`** creates an element for an XML document.
- **`Document`** method **`createTextNode`** specifies the text for an **`Element`**.
- **`Element`** method **`appendChild`** appends a node to an **`Element`** as a child of that **`Element`**.
- An XML document can be transformed into an XHTML document using an XSL style sheet.
- **`ServletRequest`** method **`getRequestDispatcher`** returns a **`RequestDispatcher`** object that can **`forward`** requests to other resources or **`include`** other resources as part of the current servlet's response.
- When **`RequestDispatcher`** method **`forward`** is called, processing of the request by the current servlet terminates.
- **`RequestDispatcher`** objects can be obtained with method **`getRequestDispatcher`** from an object that implements interface **`ServletRequest`** or from the **`ServletContext`** with methods **`getRequestDispatcher`** or **`getNamedDispatcher`**.
- **`ServletContext`** method **`getNamedDispatcher`** receives the name of a servlet as an argument, then searches the **`ServletContext`** for a servlet by that name. If no such servlet is found, this method returns **`null`**.
- Both the **`ServletRequest`** and the **`ServletContext`** **`getRequestDispatcher`** methods simply return the content of the specified path if the path does not represent a servlet.
- Before the XML and XSL capabilities can be used, you must download and install Sun's Java API for XML Parsing (JAXP) version 1.1 from **`java.sun.com/xml/download.htm`**.
- The root directory of JAXP (normally called **`jaxp-1.1`**) contains three JAR files that are required for compiling and running programs that use JAXP–**`crimson.jar`**, **`jaxp.jar`** and **`xalan.jar`**. These must be added to the Java extension mechanism for your Java 2 Standard Edition installation.
- JAXP 1.1 is part of the forthcoming J2EE 1.3 reference implementation.
- Creating a Document Object Model (DOM) tree from an XML document requires a **`DocumentBuilder`** parser object. **`DocumentBuilder`** objects are obtained from a **`DocumentBuilderFactory`**.
- Classes **`Document`** and **`Element`** are located in package **`org.w3c.dom`**.
- A **`DOMSource`** represents an XML document in an XSL transformation. A **`StreamSource`** can be used to read a stream of bytes that represent an XSL file.
- A **`StreamResult`** specifies the **`PrintWriter`** to which the results of the XSL transformation are written.
- **`TransformerFactory`** **`static`** method **`newInstance`** creates a **`TransformerFactory`** object. This object enables the program to obtain a **`Transformer`** object that applies the XSL transformation.
- **`TransformerFactory`** method **`newTransformer`** receives a **`StreamSource`** argument representing the XSL that will be applied to an XML document.
- **`Transformer`** method **`transform`** performs an XSL transformation on the given **`DOMSource`** object and writes the result to the given **`StreamResult`** object.
- A **`TransformerException`** is thrown if a problem occurs when creating the **`TransformerFactory`**, creating the **`Transformer`** or performing the transformation.
- **`HttpSession`** method **`invalidate`** discards the session object for the current client.
- Before deploying an Enterprise Java application, you must configure your data sources and other resources, so the J2EE server can register those resources with the naming server. This enables the application to use JNDI to locate the resources at execution time.

advjhtp1_11.fm Page 583 Saturday, June 9, 2001 11:04 AM

- J2EE comes with Cloudscape—a pure-Java database application from Informix Software.
- To configure a Cloudscape data source, you must modify the J2EE default configuration file **`default.properties`** in the J2EE installation's **`config`** directory. Below the comment **`JDBC URL Examples`** is a line that begins with **`jdbc.datasources`**. Append the following text (in which *dataSource* represents your data source name) to this line:

  `jdbc/`*dataSource*`|jdbc:cloudscape:rmi:`*dataSource*`;create=true`

- Each database server typically has its own URL format that enables an application to interact with databases hosted on that database server.
- You must launch the Cloudscape database server and the J2EE server before you can deploy and execute an application.
- To ensure that the J2EE server communicates properly with the Cloudscape server (or any other database server), always launch the database server before the J2EE server.
- Always shut down the J2EE server before the Cloudscape database server to ensure that the J2EE server does not attempt to communicate with the Cloudscape database server after the database server has been shut down.
- The J2EE reference implementation comes with a graphical application called the **Application Deployment Tool** that helps you deploy Enterprise Java applications.
- The **Application Deployment Tool** is nice in that it writes the deployment descriptor files for you and automatically archives the Web application's components. The tool places all Web application components and auxiliary files for a particular application into a single Enterprise Application Archive (EAR) file. This file contains deployment descriptor information, WAR files with Web application components and additional information that is discussed later in the book.
- When adding a class file for a class in a package to an application with the **Application Deployment Tool**, it is imperative that the files be added and maintained in their full package directory structure or as part of a JAR file that contains the full package directory structure.
- Not including the full package directory structure for a class in a package will prevent the application from loading the class and prevent the application from executing properly.
- The Web context for an application is the part of the URL that enables the server to determine which application to execute when the server receives a request from a client.
- The J2EE server uses port 8000, rather than the port 8080 used by Tomcat.
- When deploying an Enterprise Java application on a production application server, it is typically not necessary to specify a port number in the URL when accessing the application.
- Part of deploying an application is to specify the resource references for the components in the application.
- Each resource reference has a corresponding JNDI name that is used by the deployment tool to register the resource with the Java Naming and Directory Service. This enables the resource to be located by the application at execution time.

TERMINOLOGY

`appendChild` method of **`Element`**
Application Deployment Tool
bookstore case study
Cascading Style Sheets (CSS)
client tier
Cloudscape
component environment entries

advjhtp1_11.fm Page 584 Saturday, June 9, 2001 11:04 AM

configure a data source
connect to a database
create a Web component
crimson.jar
CSS attribute
database resource
database tier
DataSource interface
default.properties
deploy an application
distributed Web application
Document interface
Document Object Model (DOM)
DocumentBuilder class
DocumentBuilderFactory class
DOMSource class
dynamic XHTML document
Element interface
Enterprise Application Archive (EAR) file
external resource
forward a client request
forward method of **RequestDispatcher**
generate XHTML
getRequestDispatcher method of **ServletRequest**
getServletContext method of **ServletContext**
include content from a resource
include method of **RequestDispatcher**
InitialContext class
invalidate method of **HttpSession**
J2EE **config** directory
Java 2 Enterprise Edition 1.2.1 reference implementation
Java API for XML Parsing (JAXP)
Java Naming and Directory Interface (JNDI)
javax.naming package
javax.xml.parsers package
javax.xml.transform package
javax.xml.transform.dom package
javax.xml.transform.stream package
jdbc.datasources J2EE configuration property
jdbc:cloudscape:rmi:books JDBC URL
JNDI name
library JAR file
locate a naming service
lookup method of **InitialContext**
name lookup
name resolution
naming service
org.w3c.dom package
org.xml.sax package
ParserConfigurationException class
register a data source with a naming server
RequestDispatcher interface
server tier
ServletContext interface
shopping cart
StreamResult class
StreamSource class

advjhtp1_11.fm Page 585 Saturday, June 9, 2001 11:04 AM

style sheet
`transform` method of **`Transformer`**
`Transformer` class
`TransformerException` class
`TransformerFactory` class
Web component
Web context
welcome file
XML
XSL transformation

SELF-REVIEW EXERCISES

11.1 Fill in the blanks in each of the following:

a) A three-tier, distributed Web application consists of _____, _____ and _____ tiers.
b) The _____ is the default document sent as the response to a client when the client initially interacts with a J2EE application.
c) The _____ enables Enterprise Java application components to access information and resources (such as databases) that are external to an application.
d) An _____ object provides access to the application's naming environment.
e) A **`RequestDispatcher`** object can _____ requests to other resources or _____ other resources as part of the current servlet's response.
f) Sun's _____ provides XML and XSL capabilities in a Java program.
g) Method _____ of interface _____ discards the session object for the current client.
h) The _____ for an application is the part of the URL that enables the server to determine which application to execute when the server receives a request from a client.
i) An Enterprise Java application container must provide a _____ that implements JNDI and enables the components executing in that container to perform name lookups to locate resources.
j) The J2EE reference implementation comes with a graphical application called the _____ that helps you deploy Enterprise Java applications.

11.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The J2EE server uses port 8080 to await client requests.
b) When deploying applications with the J2EE server, you can launch the Cloudscape and J2EE servers in any order.
c) **`InitialContext`** method **`lookup`** locates a resource with a JNDI name.
d) Method **`lookup`** returns a **`Connection`** object representing the connection to the database.
e) The Java 2 Enterprise Edition 1.2.1 reference implementation includes the Apache Tomcat JSP and servlet container.
f) When **`RequestDispatcher`** method **`forward`** is called, processing of the request by the current servlet is temporarily suspended to wait for a response from the resource to which the request is forwarded.
g) Both the **`ServletRequest`** and the **`ServletContext`** **`getRequestDispatcher`** methods throw exceptions if the argument to **`getRequestDispatcher`** is not a servlet.
h) Each resource reference has a corresponding JNDI name that is used by the deployment tool to register the resource with the Java Naming and Directory Service.

advjhtp1_11.fm Page 586 Saturday, June 9, 2001 11:04 AM

i) If you do not configure your data sources and other resources before deploying an Enterprise Java application, the J2EE server will search the application to determine the resources used and register those resources with the naming server.
j) Not including the full package directory structure for a class in a package will prevent the application from loading the class and prevent the application from executing properly.

ANSWERS TO SELF-REVIEW EXERCISES

11.1 a) client, server, database. b) welcome file. c) Java Naming and Directory Interface (JNDI). d) **`InitialContext`**. e) **`forward`**, **`include`**. f) Java API for XML Parsing (JAXP). g) **`HttpSession`**, **`invalidate`**. h) Web context. i) naming service. j) **Application Deployment Tool**.

11.1
a) False. Port 8080 is the default port for the Tomcat server. The J2EE server uses port 8000.
b) False. To ensure that the J2EE server communicates properly with the Cloudscape server (or any other database server), the database server must be launched before the J2EE server.
c) True.
d) False. Method **`lookup`** returns a **`DataSource`** object that can be used to obtain a **`Connection`**.
e) True.
f) False. When **`RequestDispatcher`** method **`forward`** is called, processing of the request by the current servlet terminates.
g) False. Both the **`ServletRequest`** and the **`ServletContext getRequestDispatcher`** methods simply return the content of the specified path if the path does not represent a servlet.
h) True.
i) False. Before deploying an Enterprise Java application, you must configure your data sources and other resources, so the J2EE server can register those resources with the naming server. Otherwise, exceptions will occur when the application attempts to access the resources.
j) True.

EXERCISES

11.1 Modify the bookstore case study to enable the client to change the quantity of an item currently in the shopping cart. Display the quantity in an **`input form`** element of type **`text`**. Provide the user with a **`submit`** button with the value **`Update Cart`** that enables the user to submit the form to a servlet that updates the quantity of the items in the cart. The servlet should forward the request to **`viewCart.jsp`** so the user can see the updated cart contents. Redeploy the bookstore application and test the update capability.

11.2 Enhance the bookstore case study's **`TitlesBean`** to obtain author information from the **`books`** database. Incorporate that author data into the **`BookBean`** class and display the author information as part of the Web page users see when they select a book and view that book's information.

11.3 Add server-side form validation to the order form in the bookstore case study. Check that the credit-card expiration date is after today's date. Make all the fields in the form required fields. When the user does not supply data for all fields, return an XHTML document containing the order form. Any fields in which the user previously entered data should contain that data. For this exercise, replace the **`order.html`** document with a JSP that generates the form dynamically.

advjhtp1_11.fm Page 587 Saturday, June 9, 2001 11:04 AM

11.4 Create an **`order`** table and an **`orderItems`** table in the **`books`** database to store orders placed by customers. The order table should store an **`orderID`**, an **`orderDate`** and the **`email`** address of the customer who placed the order. [*Note*: you will need to modify the **`form`** in Exercise 11.4 to include the customer's email address]. The **`orderItems`** table should store the **`orderID`**, **`ISBN`**, **`price`** and **`quantity`** of each book in the order. Modify **`process.jsp`** so that it stores the order information in the **`order`** and **`orderItems`** tables.

11.5 Create a JSP that enables client to view their order history. Integrate this JSP into the bookstore case study.

11.6 Create and deploy a single application that allows a user to test all the JSP examples in Chapter 10. The application should have a welcome file that is an XHTML document containing links to each of the examples in Chapter 10.

11.7 Create and deploy a single application that allows a user to test all the servlets in Chapter 9. The application should have a welcome file that is an XHTML document containing links to each of the examples in Chapter 9.