FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

The Complete Reference

Chapter 11

Java ServerPages

A Java ServerPage (JSP) is a server-side program that is similar in design and functionality to a Java servlet, which is described in the previous chapter. A JSP is called by a client to provide a web service, the nature of which depends on the J2EE application. A JSP processes the request by using logic built into the JSP or by calling other web components built using Java servlet technology or Enterprise JavaBean technology, or created using other technologies. Once the request is processed, the JSP responds by sending the results to the client.

However, a JSP differs from a Java servlet in the way in which the JSP is written. As you'll recall from Chapter 9, a Java servlet is written using the Java programming language and responses are encoded as an output String object that is passed to the println() method. The output String object is formatted in HTML, XML, or whatever formats are required by the client.

In contrast, JSP is written in HTML, XML, or in the client's format that is interspersed with scripting elements, directives, and actions comprised of Java programming language and JSP syntax. In this chapter you'll learn how to create a JSP that can be used as a middle-level program between clients and web services.

JSP

A JSP is simpler to create than a Java servlet because a JSP is written in HTML rather than with the Java programming language. This means that the JSP isn't cluttered with many println() methods as found in a Java servlet. However, a JSP offers basically the same features found in a Java servlet because a JSP is converted to a Java servlet the first time that a client requests the JSP.

There are three methods that are automatically called when a JSP is requested and when the JSP terminates normally. These are the jspInt() method, the jspDestroy() method, and the service() method. These methods can be overridden, although the jspInt() method and jspDestroy() methods are commonly overridden in a JSP to provide customized functionality when the JSP is called and terminates.

The jspInt() method is identical the init() method in a Java servlet and in an applet. The jspInt() method is called first when the JSP is requested and is used to initialize objects and variables that are used throughout the life of the JSP.

The jspDestroy() method is identical to the destroy() method in a Java servlet. The destroy() method is automatically called when the JSP terminates normally. It isn't called if the JSP abruptly terminates such as when the server crashes. The destroy() method is used for cleanup where resources used during the execution of the JSP are released, such as disconnecting from a database.

The service() method is automatically called and retrieves a connection to HTTP.

Installation

Once a JSP is created, you place the JSP in the same directory as HTML pages. This differs from a Java servlet, which must be placed in a particular directory that is included in the CLASSPATH. You don't need to set the CLASSPATH to reference a JSP.

However, there are three factors that you must address when installing a JSP. First, web services called by a JSP must be installed properly. For example, a Java servlet called by a JSP must be placed in the designated directory for Java servlets and referenced on the CLASSPATH. The development environment used to create the J2EE application determines the designated directory.

The second factor to be addressed is to avoid placing the JSP in the WEB-INF or META-INF directories. The development environment prohibits this. The last factor is that the directory name used to store a JSP mustn't have the same name as the prefix of the URL of the web application.

JSP Tags

A JSP program consists of a combination of HTML tags and JSP tags. JSP tags define Java code that is to be executed before the output of the JSP program is sent to the browser.

A JSP tag begins with a <%, which is followed by Java code, and ends with %>. There is also an Extendable Markup Language (XML) version of JSP tags, which are formatted as <jsp:TagID> </JSP:TagID>.

JSP tags are embedded into the HTML component of a JSP program and are processed by a JSP virtual engine such as Tomcat, which is discussed later in this chapter. Tomcat reads the JSP program whenever the program is called by a browser and resolves JSP tags, then sends the HTML tags and related information to the browser.

Java code associated with JSP tags in the JSP program is executed when encountered by Tomcat, and the result of that process is sent to the browser. The browser knows how to display the result because the JSP tag is enclosed within an open and closed HTML tag. You'll see how this works in the next section.

There are five types of JSP tags that you'll use in a JSP program. These are as follows:

- **Comment tag** A comment tag opens with <%-- and closes with --%>, and is followed by a comment that usually describes the functionality of statements that follow the comment tag.
- **Declaration statement tags** A declaration statement tag opens with <%! and is followed by a Java declaration statement(s) that define variables, objects, and methods that are available to other components of the JSP program.
- **Directive tags** A directive tag opens with <%@ and commands the JSP virtual engine to perform a specific task, such as importing a Java package required by objects and methods used in a declaration statement. The directive tag closes with %>. There are three commonly used directives. These are import, include, and taglib. The import tag is used to import Java packages into the JSP program. The include tag inserts a specified file into the JSP program replacing the include tag. The taglib tag specifies a file that contains a tag library. Here are examples of each tag. The first tag imports the java.sql package. The next tag includes

the books.html file located in the keogh directory. And the last tag loads the myTags.tld library.

```
<%@ page import=" import java.sql.*"; %>
<%@ include file="keogh\books.html" %>
<%@ taglib uri="myTags.tld" %>
```

- **Expression tags** An expression tag opens with <%= and is used for an expression statement whose result replaces the expression tag when the JSP virtual engine resolves JSP tags. An expression tags closes with %>.
- **Scriptlet tags** A scriptlet tag opens with <% and contains commonly used Java control statements and loops. A scriptlet tag closes with %>.

Variables and Objects

You can declare Java variables and objects that are used in a JSP program by using the same coding technique as used to declare them in Java. However, the declaration statement must appear as a JSP tag within the JSP program before the variable or object is used in the program.

Listing 11-1 shows a simple JSP program that declares and uses a variable. In this example, the program declares an int called age and initializes the variable with the number 29. The declaration statement is placed within JSP tag.

You'll notice that this JSP tag begins with <%!. This tells the JSP virtual engine to make statements contained in the tag available to other JSP tags in the program. You'll need to do this nearly every time you declare variables or objects in your program unless they are only to be used within the JSP tag where they are declared.

The variable age is used in an expression tag that is embedded within the HTML paragraph tag <P>. A JSP expression tag begins with <%=, which is followed by the expression.

The JSP virtual engine resolves the JSP expression before sending the output of the JSP program to the browser. That is, the JSP tag <%=age%> is replaced with the number 29; afterwards, the HTML paragraph tag and related information is sent to the browser.

Listing 11-1 Declaring and using a variable.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! int age=29; %>
    <P> Your age is: <%=age%> </P>
  </BODY>
</HTML>
```

Any Java declaration statement can be used in a JSP tag similar to how those statements are used in a Java program. You are able to place multiple statements within a JSP tag by extending the close JSP tag to another line in the JSP program. This is illustrated in Listing 11-2 where three variables are declared.

Listing 11-2 Declaring multiple variables within a single JSP tag.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! int age=29;
      float salary;
      int empnumber;
    %>
  </BODY>
</HTML>
```

Besides variables, you are also able to declare objects, arrays, and Java collections within a JSP tag using techniques similar to those used in a Java program. Listing 11-3 shows how to declare an object and an array.

In Listing 11-3, the JSP program creates three String objects, the first two declarations implicitly allocate memory, and the third declaration explicitly allocates memory. In addition, this JSP program creates arrays and a Vector.

Listing 11-3 Declaring objects and arrays within a single JSP tag.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String Name;
      String [ ] Telephone = {"201-555-1212", "201-555-4433"};
      String Company = new String();
      Vector Assignments = new Vector();
      int[ ]  Grade = {100,82,93};
    %>
  </BODY>
</HTML>
```

Methods

JSP offers the same versatility that you have with Java programs, such as defining methods that are local to the JSP program. A method is defined similar to how a method is defined in a Java program except the method definition is placed within a JSP tag.

You can call the method from within the JSP tag once the method is defined. Listing 11-4 illustrates how this is done. In this example, the method is passed a student's grade and then applies a curve before returning the curved grade.

The method is called from within an HTML paragraph tag in this program, although any appropriate tag can be used to call the method. Technically, the method is called from within the JSP tag that is enclosed within the HTML paragraph tag.

The JSP tag that calls the method must be a JSP expression tag, which begins with <%=. You'll notice that the method call is identical to the way a method is called within a Java program. The JSP virtual engine resolves the JSP tag that calls the method by replacing the JSP tag with the results returned by the method, which is then passed along to the browser that called the JSP program.

Listing 11-4 Defining and calling a method.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! boolean curve (int grade)
        {
         return 10 + grade;
        }
    %>
    <P> Your curved grade is: <%=curve(80)%> </P>
  </BODY>
</HTML>
```

A JSP program is capable of handling practically any kind of method that you normally use in a Java program. For example, Listing 11-5 shows how to define and use an overloaded method.

Both methods are defined in the same JSP tag, although each follows Java syntax structure for defining a method. One method uses a default value for the curve, while the overload method enables the statement that calls the method to provide the value of the curve.

Once again, these methods are called from an embedded JSP tag placed inside two HTML paragraph tags.

sting 11-5 Defining and calling a method and an overloaded method.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! boolean curve (int grade)
        {
         return 10 + grade;
        }
        boolean curve (int grade,int curveValue)
        {
          return curveValue + grade;
        }
   %>
   <P> Your curved grade is: <%=curve(80, 10)%> </P>
   <P> Your curved grade is: <%=curve(70)%> </P>
  </BODY>
</HTML>
```


Control Statements

One of the most powerful features available in JSP is the ability to change the flow of the program to truly create dynamic content for a web page based on conditions received from the browser.

There are two control statements used to change the flow of a JSP program. These are the if statement and the switch statement, both of which are also used to direct the flow of a Java program.

The if statement evaluates a condition statement to determine if one or more lines of code are to be executed or skipped (as you probably remember from when you learned Java). Similarly, a switch statement compares a value with one or more other values associated with a case statement. The code segment that is associated with the matching case statement is executed. Code segments associated with other case statements are ignored.

The power of these controls comes from the fact that the code segment that is executed or skipped can consist of HTML tags or a combination of HTML tags and JSP tags. That is, these code segments don't need to be only Java statements or Java tags.

Listing 11-6 shows how to intertwine HTML tags and JSP tags to alter the flow of the JSP program. You'll notice that this program is confusing to read because the if statement and the switch statement are broken into several JSP tags. This is necessary because HTML tags are interspersed within these statements.

The if statement requires three JSP tags. The first contains the beginning of the if statement, including the conditional expression. The second contains the else statement, and the third has the closed French brace used to terminate the else block.

Two HTML paragraph tags contain information that the browser displays, depending on the evaluation of the conditional expression in the if statement. Only one of the HTML paragraph tags and related information are sent to the browser.

The switch statement also is divided into several JSP tags because each case statement requires an HTML paragraph tag and related information. And, as with the if statement, only one HTML paragraph tag and related information associated with a case statement that matches the switch value are returned to the browser.

Listing 11-6 contains simple examples of the if statement and switch statement. You can create more complex statements, such as nesting if statements, by using techniques illustrated in this example. Any flow control statements that you use in Java can also be incorporated into a JSP program. However, you must be careful to separate JSP tags from HTML tags and information that will be executed when the program's flow changes.

Listing 11-6 Using an if statement and a switch statement to determine which HTML tags and information are to be sent to the browser.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! int grade=70;%>
    <% if (grade > 69) { %>
         <P> You passed! </P>
    <% } else { %>
    <P> Better luck next time. </P>
    <% } %>
    <% switch (grade) {
         case 90 : %>
           <P> Your final grade is a A </P>
    <%     break; %>
         case 80 : %>
           <P> Your final grade is a B </P>
    <%     break;
         case 70 : %>
           <P> Your final grade is a C </P>
    <%     break;
         case 60 : %>
           <P> Your final grade is an F </P>
   <%      break;
       }
    %>
  </BODY>
</HTML>
```

Loops

JSP loops are nearly identical to loops that you use in your Java program except you can repeat HTML tags and related information multiple times within your JSP program without having to enter the additional HTML tags.

There are three kinds of loops commonly used in a JSP program. These are the for loop, the while loop, and the do...while loop. The for loop repeats, usually a specified number of times, although you can create an endless for loop, which you no doubt learned when you were introduced to Java.

The while loop executes continually as long as a specified condition remains true. However, the while loop may not execute because the condition may never be true. In contrast, the do...while loop executes at least once; afterwards, the conditional expression in the do...while loop is evaluated to determine if the loop should be executed another time.

Loops play an important role in JSP database programs because loops are used to populate HTML tables with data in the result set. However, Listing 11-7 shows a similar routine used to populate three HTML tables with values assigned to an array.

All the tables appear the same, although a different loop is used to create each table. The JSP program initially declares and initializes an array and an integer, and then begins to create the first table.

There are two rows in each table. The first row contains three column headings that are hard coded into the program. The second row also contains three columns each of which is a value of an element of the array.

The first table is created using the for loop. The opening table row tag <TR> is entered into the program before the for loop begins. This is because the for loop is only populating columns and not rows.

A pair of HTML table data cell tags <TD> are placed inside the for loop along with a JSP tag that contains an element of the array. The JSP tag resolves to the value of the array element by the JSP virtual program.

The close table row </TR> tag and the close </TABLE> tag are inserted into the program following the French brace that closes the for loop block. These tags terminate the construction of the table.

A similar process is used to create the other two tables, except the while loop and the do...while loop are used in place of the for loop.

Listing 11-7 Using the for loop, while loop, and the do...while loop to load HTML tables.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
```

```
<%! int[ ]  Grade = {100,82,93};
    int x=0;
%>
<TABLE>
  <TR>
    <TD>First</TD>
    <TD>Second</TD>
    <TD>Third</TD>
  </TR>
  <TR>
    <% for (int 1; i<3; i++) { %>
        <TD><%=Grade[i]%> </TD>
    <%   } %>
  </TR>
</TABLE>
<TABLE>
  <TR>
    <TD>First</TD>
    <TD>Second</TD>
    <TD>Third</TD>
  </TR>
  <TR>
    <% while (x<3){ %>
        <TD><%=Grade[x]%> </TD>
        <%    x++;
      } %>
  </TR>
</TABLE>
<TABLE>
  <TR>
   <TD>First</TD>
   <TD>Second</TD>
   <TD>Third</TD>
  </TR>
  <TR>
    <% x=0;
       do{ %>
           <TD><%=Grade[x]%></TD>
        <%x++;
        } while (x<3) %>
```

```
        </TR>
      </TABLE>
    </BODY>
  </HTML>
```

Tomcat

JSP programs are executed by a JSP Virtual Machine that runs on a web server. Therefore, you'll need to have access to a JSP Virtual Machine to run your JSP program. Alternatively, you can use an integrated development environment such as JBuilder that has a built-in JSP Virtual Machine or you can download and install a JSP Virtual Machine.

One of the most popular JSP Virtual Machines is Tomcat, and it is downloadable at no charge from the Apache web site. Apache is also a popular web server that you can also download at no cost.

You'll also need to have the Java Development Kit (JDK) installed on your computer, which you probably installed when you learned Java programming. You can download the JDK at no charge from the www.sun.com web site.

Here's what you need to do to download and install Tomcat:

1. Connect to jakarta.apache.org.
2. Select Download.
3. Select Binaries to display the Binary Download page.
4. Create a folder from the root directory called tomcat.
5. Download the latest release of jakarta-tomcat.zip to the tomcat folder.
6. Unzip jakarta-tomcat.zip. You can download a demo copy of WinZip from www.winzip.com if you don't have a zip/unzip program installed on your computer.
7. The extraction process should create the following folders in the tomcat directory: bin, conf, doc, lib src, and webapps.
8. Use a text editor such as Notepad and edit the JAVA_HOME variable in the tomcat.bat file, which is located in the \tomcat\bin folder. Make sure the JAVA_HOME variable is assigned the path where the JDK is installed on your computer.
9. Open a DOS window and type **\tomcat\bin\tomcat** to start Tomcat.
10. Open your browser. Enter **http://localhost:8080**. The Tomcat home page is displayed on the screen verifying that Tomcat is running.

Request String

The browser generates a user request string whenever the Submit button is selected. The user request string consists of the URL and the query string, as you learned at the beginning of this chapter. Here's a typical request string:

```
http://www.jimkeogh.com/jsp/myprogram.jsp?fname="Bob"&lname="Smith"
```

Your program needs to parse the query string to extract values of fields that are to be processed by your program. You can parse the query string by using methods of the JSP request object.

The getParameter(Name) is the method used to parse a value of a specific field. The getParameter() method requires an argument, which is the name of the field whose value you want to retrieve.

Let's say that you want to retrieve the value of the fname field and the value of the lname field in the previous request string. Here are the statements that you'll need in your JSP program:

```
<%! String Firstname = request.getParameter(fname);
    String Lastname = request.getParameter(lname);
%>
```

In the previous example, the first statement used the getParameter() method to copy the value of the fname from the request string and assign that value to the Firstname object. Likewise, the second statement performs a similar function, but using the value of the lname from the request string. You can use request string values throughout your program once the values are assigned to variables in your JSP program.

There are four predefined implicit objects that are in every JSP program. These are request, response, session, and out. The previous example used the request object's getParameter() method to retrieve elements of the request string. The request object is an instance of the HttpServletRequest (see Chapter 9). The response object is an instance of HttpServletResponse, and the session object is an instance of HttpSession. Both of these are described in detail in Chapter 9. The out object is an instance of the JspWriter that is used to send a response to the client.

Copying a value from a multivalued field such as a selection list field can be tricky since there are multiple instances of the field name, each with a different value. However, you can easily handle multivalued fields by using the getParameterValues() method.

The getParameterValues() method is designed to return multiple values from the field specified as the argument to the getParameterValues(). Here is how the getParameterValues() is implemented in a JSP program.

In this example, we're retrieving the selection list field shown in Listing 11-8. The name of the selection list field is EMAILADDRESS, the values of which are copied into an array of String objects called EMAIL. Elements of the array are then displayed in JSP expression tags.

Listing 11-8 Selecting a listing of fields.

```
<%! String [ ] EMAIL = request.getParameterValues("EMAILADDRESS ") ; %>
<P> <%= EMAIL [0]%> </P>
<P> <%= EMAIL [1]%> </P>
```

You can parse field names by using the getParameterNames() method. This method returns an enumeration of String objects that contains the field names in the request string. You can use the enumeration extracting methods that you learned in Java to copy field names to variables within your program.

The Quick Reference Guide in Chapter 9 contains a list of other commonly used HttpServletRequest class methods that you'll find useful when working with parameters.

Parsing Other Information

The request string sent to the JSP by the browser is divided into two general components that are separated by the question mark. The URL component appears to the left of the question mark and the query string is to the right of the question mark.

In the previous section you learned how to parse components of the query string, which are field names and values using request object methods. These are similar to the method used to parse URL information.

The URL is divided into four parts, beginning with the protocol. The protocol defines the rules that are used to transfer the request string from the browser to the JSP program. Three of the more commonly used protocols are HTTP, HTTPS (the secured version of HTTP), and FTP, which is a file transfer protocol.

Next is the host and port combination. The host is the Internet Protocol (IP) address or name of the server that contains the JSP program. The port number is the port that the host monitors. Usually the port is excluded from the request string whenever HTTP is used because the assumption is the host is monitoring port 80. Following the host and port is the virtual path of the JSP program. The server maps the virtual path to the physical path.

Here's a typical URL. The http is the protocol. The host is www.jimkeogh.com. There isn't a port because the browser assumes that the server is monitoring port 80. The virtual path is /jsp/myprogram.jsp.

```
http://www.jimkeogh.com/jsp/myprogram.jsp
```

User Sessions

A JSP program must be able to track a session as a client moves between HTML pages and JSP programs as discussed in Chapter 9. There are three commonly used methods to track a session. These are by using a hidden field, by using a cookie, or by using a JavaBean, which is discussed in the next chapter.

A hidden field is a field in an HTML form whose value isn't displayed on the HTML page, as you learned in Chapter 8. You can assign a value to a hidden field in a JSP program before the program sends the dynamic HTML page to the browser.

Let's say that your JSP database system displays a dynamic login screen. The browser sends the user ID and password to the JSP program when the Submit button is selected where these parameters are parsed and stored into two memory variables (see the "Request String" section).

The JSP program then validates the login information and generates another dynamic HTML page once the user ID and password are approved. The new dynamically built HTML page contains a form that contains a hidden field, among other fields. And the user ID is assigned as the value to the hidden field.

When the person selects the Submit button on the new HTML page, the user ID stored in the hidden field and information in other fields on the form are sent by the browser to another JSP program for processing.

This cycle continues where the JSP program processing the request string receives the user ID as a parameter and then passes the user ID to the next dynamically built HTML page as a hidden field. In this way, each HTML page and subsequent JSP program has access to the user ID and therefore can track the session.

The Quick Reference Guide in Chapter 9 contains a list of other commonly used HttpSession class methods that you'll find useful when working with a session.

Cookies

As you learned in Chapter 9, a cookie is a small piece of information created by a JSP program that is stored on the client's hard disk by the browser. Cookies are used to store various kinds of information, such as user preferences and an ID that tracks a session with a JSP database system.

You can create and read a cookie by using methods of the Cookie class and the response object as illustrated in Listing 11-9 and in Listing 11-10. Listing 11-9 creates and writes a cookie called userID that has a value of JK1234.

The program begins by initializing the cookie name and cookie value and then passes these String objects as arguments to the constructor of a new cookie. This cookie is then passed to the addCookie() method, which causes the cookie to be written to the client's hard disk.

Listing 11-10 retrieves a cookie and sends the cookie name and cookie value to the browser, which displays these on the screen. This program begins by initializing the

MyCookieName String object to the name of the cookie that needs to be retrieved from the client's hard disk. I call the cookie userID.

Two other String objects are created to hold the name and value of the cookie read from the client. Also I created an int called found and initialized it to zero. This variable is used as a flag to indicate whether or not the userID cookie is read.

Next an array of Cookie objects called cookies is created and assigned the results of the request.getCookies() method, which reads all the cookies from the client's hard disk and assigns them to the array of Cookie objects.

The program proceeds to use the getName() and getValue() methods to retrieve the name and value from each object of the array of Cookie objects. Each time a Cookie object is read, the program compares the name of the cookie to the value of the MyCookieName String object, which is userID.

When a match is found, the program assigns the value of the current Cookie object to the MyCookieValue String object and changes the value of the found variable from 0 to 1.

After the program reads all the Cookie objects, the program evaluates the value of the found variable. If the value is 1, the program sends the value of the MyCookieName and MyCookieValue to the browser, which displays these values on the screen.

The Quick Reference Guide in Chapter 9 contains a list of other commonly used Cookie class methods that you'll find useful when working with cookies.

Listing 11-9 How to create a cookie.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String MyCookieName = "userID";
        String MyCookieValue = "JK1234";
        response.addCookie(new Cookie(MyCookieName, MyCookieValue));
    %>
  </BODY>
</HTML>
```

Listing 11-10 How to read a cookie.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String MyCookieName = "userID";
        String MyCookieValue;
        String CName, CValue;
```

```
        int found=0;
        Cookie[] cookies = request.getCookies();
        for(int i=0; i<cookies.length; i++) {
          CName  = cookies[i].getName();
          CValue = cookies[i].getValue();
          if(MyCookieName.equals(cookieNames[i])) {
            found = 1;
            MyCookieValue = cookieValue;
          }
        }
        if (found ==1) { %>
          <P> Cookie name = <%= MyCookieName %> </P>
          <P> Cookie value = <%= MyCookieValue %> </P>
      <%}%>
  </BODY>
</HTML>
```

Session Objects

A JSP database system is able to share information among JSP programs within a session by using a session object. Each time a session is created, a unique ID is assigned to the session and stored as a cookie.

The unique ID enables JSP programs to track multiple sessions simultaneously while maintaining data integrity of each session. The session ID is used to prevent the intermingling of information from clients.

In addition to the session ID, a session object is also used to store other types of information, called *attributes*. An attribute can be login information, preferences, or even purchases placed in an electronic shopping cart.

Let's say that you built a Java database system that enables customers to purchase goods online. A JSP program dynamically generates catalogue pages of available merchandise. A new catalogue page is generated each time the JSP program executes.

The customer selects merchandise from a catalogue page, then jumps to another catalogue page where additional merchandise is available for purchase. Your JSP database system must be able to temporarily store purchases made from each catalogue page; otherwise, the system is unable to execute the checkout process. This means that purchases must be accessible each time the JSP program executes.

There are several ways in which you can share purchases. You might store merchandise temporally in a table, but then you'll need to access the DBMS several times during the session, which might cause performance degradation.

A better approach is to use a session object and store information about purchases as session attributes. Session attributes can be retrieved and modified each time the JSP program runs.

Listing 11-11 illustrates how to assign information to a session attribute. In this example, the program creates and initializes two String objects. One String object is assigned the name of the attribute and the other String object is assigned a value for the attribute. Next, the program calls the setAttribute() method and passes this method the name and value of the attribute.

Listing 11-12 reads attributes. The program begins by calling the getAttributeNames() method that returns names of all the attributes as an Enumeration.

Next, the program tests whether or not the getAttributeNames() method returned any attributes. If so, statements within the while loop execute, which assigns the attribute name of the current element to the AtName String object. The AtName String object is then passed as an argument to the getAttribute() method, which returns the value of the attribute. The value is assigned to the AtValue String object. The program then sends the attribute name and value to the browser.

The Quick Reference Guide in Chapter 9 contains a list of other commonly used HttpSession class methods that you'll find useful when working with a session.

Listing 11-11 How to create a session attribute.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! String AtName = "Product";
        String AtValue = "1234";
        session.setAttribute(AtName, AtValue);
    %>
  </BODY>
</HTML>
```

Listing 11-12 How to read session attributes.

```
<HTML>
  <HEAD>
    <TITLE> JSP Programming </TITLE>
  </HEAD>
  <BODY>
    <%! Enumeration purchases = session.getAttributeNames();
        while(purchases.hasMoreElements()){
          String AtName = (String)attributeNames.nextElement();
          String AtValue = (String)session.getAttribute(AtName); %>
          <P> Attribute Name <%= AtName %> </P>
          <P> Attribute Value <%= AtValue %> </P>
     <% } %>
  </BODY>
</HTML>
```

Quick Reference Guide

Syntax	Descriptions
void _jspService(HttpServletRequest request, HttpServletResponse response)	Corresponds to the body of the JSP page.

Table 11-1. *public interface HttpJspPage extends JspPage*

Sytax	Descriptions
void jspDestroy()	Automatically invoked when a JSP page is to be destroyed.
void jspInit()	Automatically invoked when the JSP page is initialized.

Table 11-2. *public interface JspPage extends Servlet*

Syntax	Descriptions
abstract Object findAttribute(String name)	Returns the value of an attribute.
abstract Object getAttribute(String name)	Returns an object associated with the name.
abstract Object getAttribute(String name, int scope)	Returns an object associated with the name.
abstract Enumeration getAttributeNamesInScope(int scope)	Returns all attributes of a scope.
abstract int getAttributesScope(String name)	Returns the scope based on name.
abstract ExpressionEvaluator getExpressionEvaluator()	Provides programmatic access to the ExpressionEvaluator.
abstract JspWriter getOut()	Determines if out object value of a JspWriter.

Table 11-3. *public abstract class JspContext extends java.lang.Object*

Syntax	Descriptions
abstract Map peekPageScope()	Peeks at the top of the page scope stack.
abstract Map popPageScope()	Removes a page scope from the stack.
abstract void pushPageScope(Map scopeState)	Places a page scope on the stack.
abstract void removeAttribute(String name)	Uses all scopes to delete an object reference associated with a name.
abstract void removeAttribute(String name, int scope)	Delete an object by name.
abstract void setAttribute(String name, Object attribute)	Registers a name associated with an object that has page scope semantics.
abstract void setAttribute(String name, Object o, int scope)	Registers a name associated with an object and scope.

Table 11-3. *public abstract class JspContext* ***extends java.lang.Object*** (continued)

Syntax	Descriptions
abstract String getSpecificationVersion()	Returns the version number of JSP

Table 11-4. *public abstract class JspEngineInfo* ***extends Object***

Syntax	Descriptions
static JspFactory getDefaultFactory()	Returns the default JSP factory.
abstract JspEngineInfo getEngineInfo()	Returns information about the JSP engine.

Table 11-5. *public abstract class JspFactory* ***extends Object***

Syntax	Descriptions
abstract PageContext getPageContext(Servlet servlet, ServletRequest request, ServletResponse response, String errorPageURL, boolean needsSession, int buffer, boolean autoflush)	Returns an instance of a PageContext.
abstract void releasePageContext (PageContext pc)	Releases an allocated PageContext object.
static void setDefaultFactory (JspFactory deflt)	Sets the default factory.

Table 11-5. *public abstract class JspFactory* ***extends Object*** (continued)

Syntax	Descriptions
abstract void clear()	Resets a buffer.
abstract void clearBuffer()	Resets a buffer.
abstract void close()	Closes and flushes a stream.
abstract void flush()	Flushes a stream.
int getBufferSize()	Returns the size of the buffer used by the JspWriter.
abstract int getRemaining()	Returns the number of unused bytes in the buffer.
boolean isAutoFlush()	Determines if a JspWriter is autoFlushing.
abstract void newLine()	Writes a line separator.

Table 11-6. *public abstract class JspWriter* ***extends Writer***

Syntax	Descriptions
abstract void print(boolean b)	Prints a boolean value.
abstract void print(char c)	Prints a character.
abstract void print(char[] s)	Prints an array of characters.
abstract void print(double d)	Prints a double-precision floating-point number.
abstract void print(float f)	Prints a floating-point number.
abstract void print(int i)	Prints an integer.
abstract void print(long l)	Prints a long integer.
abstract void print(Object obj)	Prints an object.
abstract void print(String s)	Prints a string.
abstract void println()	Writes a line separator string to terminate a line.
abstract void println(boolean x)	Prints a boolean value with a terminated line
abstract void println(char x)	Prints a character with a terminated line.
abstract void println(char[] x)	Prints an array of characters with a terminated line
abstract void println(double x)	Prints a double-precision floating-point number with a terminated line.
abstract void println(float x)	Prints a floating-point number with a terminated line.
abstract void println(int x)	Prints an integer with a terminated line.
abstract void println(long x)	Prints a long integer with a terminated line.
abstract void println(Object x)	Prints an Object with a terminated line.
abstract void println(String x)	Prints a String with a terminated line.

Table 11-6. *public abstract class JspWriter* ***extends Writer*** (continued)

Syntax	Descriptions
abstract void forward(String relativeUrlPath)	Redirects the ServletRequest and ServletResponse to another active component in the application.
abstract Exception getException()	Returns the value of the exception object.
abstract Object getPage()	Returns the value of the page object.
abstract ServletRequest getRequest()	Returns the value of the response object.
abstract ServletResponse getResponse()	Returns the value of the response object.
abstract ServletConfig getServletConfig()	Returns the instance of the ServletConfig.
abstract ServletContext getServletContext()	Returns the instance of the ServletContext.
abstract HttpSession getSession()	Returns the value of the session object.
abstract void handlePageException (Exception e)	Redirects an unhandled page level exception to an error page.
abstract void handlePageException (Throwable t)	Makes an unhandled page level exception Throwable.
abstract void **include**(String relativeUrlPath)	Processes the resource as part of the current ServletRequest.
abstract void initialize(Servlet servlet, ServletRequest request, ServletResponse response, String errorPageURL, boolean needsSession, int bufferSize, boolean autoFlush)	Initializes an uninitialized PageContext.
JspWriter popBody()	Updates the page scope "out" attribute of the PageContext; and returns the previous JspWriter "out" saved by the matching pushBody().
BodyContent pushBody()	Saves the current "out" JspWriter; updates the page scope "out" attribute of the PageContext; and returns a new BodyContent object.
abstract void release()	Resets the internal state of a PageContext, for potential reuse.

Table 11-7. *public abstract class PageContext* ***extends JspContext***

Syntax	Descriptions
Object evaluate(String attributeName, String expression, Class expectedType, Tag tag, PageContext pageContext)	Evaluates the expression contained in a request.
String validate(String attributeName, String expression)	Validates an expression at translation time.

Table 11-8. *public interface* ***ExpressionEvaluator***

Syntax	Descriptions
Object resolveVariable(String pName, Object pContext)	Resolves a variable within the given context.

Table 11-9. *public interface* ***VariableResolver***

Syntax	Descriptions
void doInitBody()	Prepares to evaluate the body.
void setBodyContent(BodyContent b)	Sets the bodyContent property.

Table 11-10. *public interface BodyTag extends IterationTag*

Syntax	Descriptions
void setDynamicAttribute(String uri, String localName, Object value)	Sets a dynamic attribute that is not declared in the Tag Library Descriptor.

Table 11-11. *public interface DynamicAttributes*

Syntax	Descriptions
int doAfterBody()	Processes the body content.

Table 11-12. *public interface IterationTag* ***extends Tag***

Syntax	Descriptions
void invoke(Writer out, Map params)	Executes the fragment and directs all output to a Writer.

Table 11-13. *public interface* ***JspFragment***

Syntax	Descriptions
int doTag()	Processes a tag.
Object getParent()	Returns the parent of a tag.
void setJspBody(JspFragment jspBody)	Sets the body of a tag as a JspFragment object.
void setJspContext(JspContext pc)	Set a page context in the protected jspContext field.
void setParent(Object parent)	Sets a parent of a tag.

Table 11-14. *public interface* ***SimpleTag***

Syntax	Descriptions
int doEndTag()	Processes an end tag.
int doStartTag()	Processes a start tag.
Tag getParent()	Returns the parent for a tag.
void release()	Instructs a Tag handler to release state.
void setPageContext(PageContext pc)	Sets a page context.
void setParent(Tag t)	Sets the parent of a tag handler.

Table 11-15. *public interface* ***Tag***

Syntax	Descriptions
void doCatch(Throwable t)	Executes when a Throwable exception happens while the body is being evaluated.
void doFinally()	Executes after doEndTag().

Table 11-16. *public interface* ***TryCatchFinally***