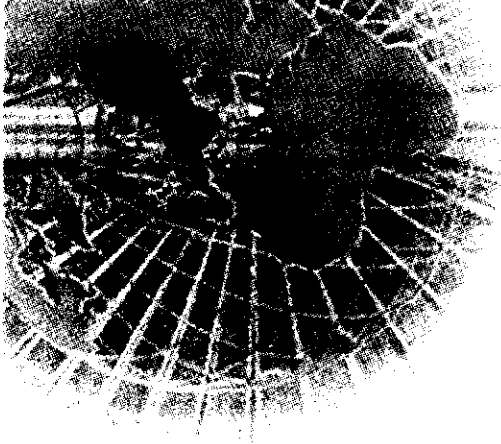

12

Working with JSP

If you need information on:	See page:
Understanding JSP	390
Describing the JSP Life Cycle	393
Creating Simple JSP Pages	396
Working with JSP Basic Tags and Implicit Objects	399
Using JavaBeans and Action Tags in JSP	412
Using the JSP Standard Tag Library (JSTL)	430
Describing JSTL Core Tags	430
Describing the JSTL SQL Tags	441
JSTL Formatting Tags	446
JSTL XML Tags	457
Implementing JSTL Tags	460

The *Chapter11, Working with Servlet Programming,* has described the importance of Servlets in Web programming. Servlets have, till now, been effectively used to create efficient Web applications. However, Servlets too are giving way to superior technology. These days, Servlets, as a popular means to create Web pages, have been replaced by better technologies, such as JavaServer Pages (**JSP**). Java Servlets have many shortcomings that have contributed to their decline. The major shortcomings are:

- **Inability to use IDEs**—You cannot use Integrated Development Environments (IDEs) to design HTML views. This increases the development time of Web applications.
- **Inability to use HTML Designer**—You cannot use HTML designer services to prepare the presentation view. Instead, you need to depend on the Java programmer to prepare this view.
- **Dependence on Web configuration**—Servlet programs depend on the Web configuration. The web.xml file is used to configure your Servlet. When you create several views by using Servlets, you have to ensure that, in the web.xml file, each Servlet is mapped to invoke the correct Servlet. This increases the complexity of your Web site because you must know which Servlet-mapping (in the web.xml file) has been used to invoke which Servlet.
- **Recurring Recompilation**—When you make changes in the code of a Servlet, you need to recompile the Servlet's source file. This is a problem when we create complex views because each time you make a change; you have to compile the modified Servlet program before generating the output.

To solve these problems, we want our HTML page, generated with the help of some IDEs, to include additional instructions wherever dynamic content is required. For example, a page with additional instructions can be parsed and resolved into a standard server-side extension (that is, a Servlet), which is understood by the Web container. If these additional instructions are tag-based, they can be directly used by Web designers to create HTML views. Moreover, tag-based instructions are easily understood by the IDEs used to design HTML views. The requirement of tag-based instruction has impelled the introduction of JSP technology.

In this chapter, we discuss what JSP is and how it is used to create dynamic Web pages. We also discuss why JSP is preferred over Servlets to create views. In addition, we look at the JSP architecture as well as give a detailed description of the various stages of the JSP life-cycle. We also create a simple JSP page. In addition, you learn about scripting and directive tags used in JSP. Moreover, the chapter also explains how to use the implicit objects. The chapter also demonstrates the implementation of JavaBeans in a JSP page. Towards the end of the chapter, the JSP standard tag library is also explained in detail.

Let's start with a brief introduction of JSP.

Understanding JSP

JSP is a standard Java extension used to simplify the creation and management of dynamic Web pages. JSPs allow to separate the dynamic content of a Web page from its static presentation content. As described earlier, programming in Servlets is very complex and requires additional files, such as web.xml and Java class files, to generate the required Web pages. JSP offers an easier approach to build dynamic Web pages. JSP documents consist of HTML tags and special tags, known as JSP tags. HTML tags are used to create static page content and JSP tags are used to add dynamic content to Web pages. JSP pages are compiled into a Java Servlet by a JSP translator. This Java Servlet is then compiled and executed to generate an output for the browser (client). Building dynamic Web pages by using JSP offers the following benefits:

- Web pages created by using JSP are portable and can be used easily across multiple platforms and Web servers without making any changes.
- Programming in JSP is easier than in Servlets because of the introduction of tag-based approach in JSP.
- JSP pages are automatically compiled by the Web server, such as Tomcat and WebLogic. Developers need not compile a JSP page when the source code of the JSP page is changed.
- One of the most important benefits of JSP is the separation of business logic from the presentation logic. The tag handler classes or JavaBeans contain business logic whereas JSP pages contain presentation logic. The separation of business and presentation logic makes an application created in JSP more secure and reusable.

Now that we are familiar with the approach to build dynamic Web pages using JSP, let's learn about the advantages of JSP over Servlets.

Advantages of JSP over Servlets

JSP provides a better server-side scripting support as compared to Servlets. Java developers can create Servlets easily, but for Web page designers, handling Java code is always a tough job. A JSP page can have HTML code, and some Java code embedded with the HTML code. In this way, creating a Web page becomes easy as the basic designing of the page is done by using the HTML code. We can use the Java code to add dynamic content to the page. The Java code is embedded by using different JSP constructs, which is discussed later in the chapter.

The following are the reasons to prefer JSP over Servlets:

- Most of the content of a Web page is static and different components need to be placed on the page in a symmetrical way. These static pages are designed by Web designers by using HTML code. Designing static pages using Servlets require a good knowledge of Java, but Web designers may not be comfortable with Java programming constructs. In addition, while using Servlets, you have to enclose HTML code in the out.println () method, which disables all IDE support for generating HTML content. Therefore, using JSP is always easier than using Servlets for the same HTML output.
- You need to recompile your Servlet for every single change in the source code of the Servlet; whereas in case of JSP, recompilation is not required since JSP are automatically handled by the Web container for any update in their code.
- Servlets cannot be accessed directly and have to be first mapped in the web.xml file, whereas a JSP page can be accessed directly as a simple HTML page.
- Both, Servlets and JSPs are server-side components used to generate dynamic HTML pages; however, JSP is preferred over Servlet as it is developed by using a simple HTML template and automatically handled by the JSP container.

These advantages of JSP make it a popular component to use in the presentation layer of any Web application.

To bring out the advantage of using JSP over Servlets, let's take an example. Let's assume that a user needs a Web page with some dynamic content (for example, current date and time).

You can create a simple Servlet to display current date and time. Listing 12.1 shows the code for the FirstServlet Servlet and its doGet() method, which generates the required output (you can find the FirstServlet.java file in the code\Java EE\Chapter 12\JSPEx folder):

Listing 12.1: The FirstServlet.java File

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;
public class FirstServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.write("<html>");
      out.write("<head>");
      out.write("<title>First JSP Page</title>");
      out.write("</head>");
      out.write("<body>");
      out.write(new Date().toString());
      out.write("<hr/>");
      out.write("<br/>Hello World!<br/>");
      out.write("</body>");
      out.write("</html>");
      out.close();
  }
}
```

This source code for your Servlet needs to be compiled, and the generated class file needs to be placed in the WEB-INF\classes folder of a Web application. In addition, a Servlet mapping is done in the deployment descriptor (web.xml) to access this Servlet.

Alternatively, you can replace this Servlet by a simple JSP page. Listing 12.2 provides the code for the JSP page to display current date and time (you can find the firstpage.jsp file in the code\Java EE\Chapter 12/JSPEx/firstpage.jsp folder on the CD):

Listing 12.2: The firstpage.jsp File

```
<%@ page import="java.util.Date;"%>
<html>
    <head>
        <title>First JSP Page</title>
    </head>
    <body>
        <%
                out.println(new Date().toString());
        %>
        <br>Hello World! </br>
    </body>
</html>
```

You can see some delimiters and objects being used here, such as <%@ %>, <% %>, and out. These JSP elements and implicit objects have been discussed in detail in the *Working with JSP Basic Tags* and *Working with the Implicit Objects* sections in this chapter. This page can be accessed as a simple HTML page and does not need to be compiled and mapped in web.xml every time the Web container accesses it. When you compare the views generated by the Servlet and JSP pages, you find that both show the same output.

We have discussed the advantages of JSP over Servlet. Next, we discuss the tag-based approach with respect to JSP.

Introducing the Tag-Based Approach

JSP is totally tag-based, which means that each piece of code in JSP is enclosed within a tag. JSP tags reduce the necessity of large amount of Java code in JSP pages by implementing the functionality of the tags into tag implementation classes. These tags help developers build dynamic pages, improve their code by reducing Java code, and separate the presentation logic from the business logic.

JSP tags have the following features:

- Reduce Web development and maintenance costs
- Help you to reduce Java code in a JSP page
- Make the task of developing Web applications easier and faster
- Enable developers to reuse Java code in multiple Web applications
- Ensure ease of work with Java Beans within JSP tags

After learning the benefits of the tag-based approach used in JSP, let's discuss the JSP architecture next.

Describing the JSP Architecture

JSP specifications demonstrate two approaches to build Web applications by using JSP. These approaches are known as JSP Model 1 and JSP Model 2 architecture, respectively. Let's look these models in detail.

The JSP Model I Architecture

In JSP Model I architecture, the Web browser directly accesses the JSP pages of a Web container. The JSP pages interact with the Web container's JavaBeans, which represent the application model. When a client sends a request from a JSP page, the response (that is, the JSP page) is sent back to the client depending on which hyperlinks are selected or which request parameters are invoked by the client request. If the generated response requires accessing a database, the JSP page uses JavaBean, which gets the required data from the database. Figure 12.1 shows the architecture of JSP Model I:

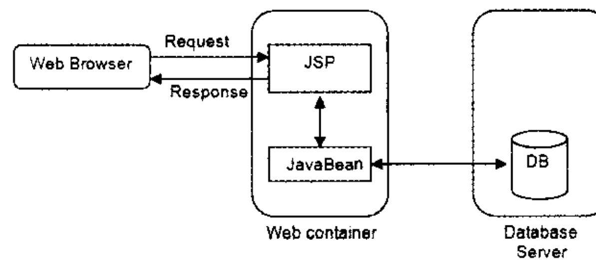

Figure 12.1: Showing JSP Model I architecture

The JSP Model II Architecture

The JSP Model II architecture uses Servlets as well, which are positioned between the Web browser and the JSP pages. The Servlet acts as a controller to dispatch requests to the next JSP view, such as the URL, based on the client request and input parameters. It also creates JavaBean instances, if they are required by the JSP page. The controller also decides which JSP page needs to be forwarded as a response, based on the client request. The JavaBean invokes the database server if data is needed from a database server. Figure 12.2 shows the architecture of JSP Model II.

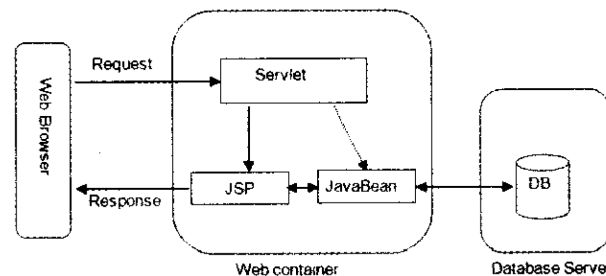

Figure 12.2: Showing JSP Model II architecture

Now that we are familiar with JSP architecture, let's learn about the JSP life-cycle in the next section.

Describing the JSP Life Cycle

Before discussing the life cycle of JSP, it must be remembered that JSP isn't executed directly. The execution of JSP follows a process in which a JSP page is first translated into its corresponding Servlet and then the source file of this Servlet is compiled into a class file. This Servlet is then loaded into memory and initialized, similar to any other Servlet. Every time a JSP is requested, its corresponding Servlet is executed. Therefore, most of the stages of the JSP life cycle are similar to that of a Servlet. The major stages of the life cycle of a JSP page are as follows:

- Page Translation
- Compilation
- Loading & Initialization
- Request Handling
- Destroying (End of service)

Let's now discuss each stage of the JSP life-cycle one by one.

The Page Translation Stage

In the page translation stage of the JSP life-cycle, the Web container translates the JSP document into an equivalent Java code—that is, a Servlet. Usually, a Servlet contains Java code with some markup language tags, such as HTML or XML. JSPs, on the other hand, consist primarily of markup language with some Java code. The objective of page translation, therefore, is to convert a document chiefly consisting of HTML/XML code, to one that has more of Java code, which can be executed by the Java Virtual Machine (JVM).

The translation of JSP is automatically done by the Web server. The translation of JSP can take place either at the time of deploying the JSP (a process generally known as JSP pre-compilation), or at the time when a request for the JSP is received for the first time.

In this stage, the Web container performs the following operations:

- Locates the requested JSP page
- Validates the syntactic correctness of the JSP page
- Interprets the standard JSP directives, actions, and custom actions used in the JSP page
- Writes the source code of the equivalent Servlet for the JSP page

All requests for a given JSP page are served by its corresponding Servlet class. This Servlet class works as a simple Servlet, similar to what we designed in *Chapter 11, Working with Servlet Programming.*

Listing 12.3 shows the Servlet generated for the firstpage JSP page:

Listing 12.3: The firstpage_jsp.java File

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.util.Date;
public final class firstpage_jsp extends org.apache.jasper.runtime.HttpJspBase
  implements org.apache.jasper.runtime.JspSourceDependent {
  private static java.util.List _jspx_dependants;
  public Object getDependants() {
  return _jspx_dependants;
  }
  public void _jspService(HttpServletRequest request,
  HttpServletResponse response)
  throws java.io.IOException, ServletException {
  JspFactory _jspxFactory = null;
  PageContext pageContext = null;
  HttpSession session = null;
  ServletContext application = null;
  ServletConfig config = null;
  JspWriter out = null;
  Object page = this;
  JspWriter _jspx_out = null;
  PageContext _jspx_page_context = null;
  try {
       _jspxFactory = JspFactory.getDefaultFactory();
       response.setContentType("text/html");
       pageContext = _jspxFactory.getPageContext(this, request, response,
       null, true, 8192, true);
       _jspx_page_context = pageContext;
       application = pageContext.getServletContext();
       config = pageContext.getServletConfig();
       session = pageContext.getSession();
       out = pageContext.getOut();
       _jspx_out = out;
       out.write("\r\n");
       out.write("<html>\r\n");
       out.write("    <head>\r\n");
       out.write("        <title>First JSP Page</title>\r\n");
       out.write("    </head>  \r\n");
       out.write("    <body>\r\n");
       out.write("        ");
       out.println(new Date().toString());
       out.write("\r\n");
```

```
            out.write("        <hr/>\r\n");
            out.write("        <br/>Hello World!<br/>\r\n");
            out.write("    </body>\r\n");
            out.write("</html>\r\n");
            out.write("\r\n");
            } catch (Throwable t) {
            if (!(t instanceof SkipPageException)){
            out = _jspx_out;
            if (out != null && out.getBufferSize() != 0)
            out.clearBuffer();
            if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
            }
        } finally {
        if (_jspxFactory != null) _jspxFactory.releasePageContext(_jspx_page_context);
            }
    }
    }
```

The Servlet created as a result of the translation of JSP has the following special characteristics:

- The name of the Servlet generated for a JSP page is the name of JSP merged with "jsp" and separated by an underscore. Therefore, for the firstpage.jsp page, we have a Servlet class with the name "firstpage_jsp" and its source code, firstpage_jsp.java, as shown in Listing 12.3.
- The generated Servlet class is a subtype of the javax.servlet.jsp.JspPage class or the javax.servlet.jsp.HttpJspPage interface, depending on the protocols you are using. HttpJspPage is most widely used, whereas JspPage is used by Web containers that support non-Http protocols.
- The generated Servlet implements the jspInit(), jspDestroy(), and _jspService() methods. You can compare them with the init (), destroy (), and service () methods of a common Servlet class, since they are used in the same manner.

The jspInit() and jspDestroy() methods are invoked once in the lifetime of JSP, whereas the _jspService() method is executed each time a JSP page is requested. All the presentation logic placed in a JSP page is enclosed in the _jspService() method. You can see how different objects, such as pageContext, application, session, and out, have been obtained and used in the _jspService() method. We will study about these implicit objects of JSP in the *Working with the Implicit Objects* section of this chapter.

The JSP page translation occurs only when necessary, that is, at some point before a JSP page has to serve its first request. This process is also performed when the JSP source code is updated and the page is redeployed. A Web container can decide when the translation should occur. The two possible instances where a translation can occur are:

- When a first-time user requests a JSP page.
- When a JSP is loaded into a Web container. This is also called JSP pre-compilation.

If there is any error in the translation of the JSP life cycle, then the container raises an exception, 500 (internal server error). If any change occurs in the JSP page, all the stages of the JSP life cycle are executed again.

The Compilation Stage

The compilation stage of the JSP life cycle follows the page translation stage. In this stage, the JSP container compiles the Java source code for the corresponding Servlet and converts it into Java byte (class) code. After the class file is generated, the container can decide to either discard the code or retain it for debugging. Generally, most containers discard the generated Java source code by default. After the compilation stage, the Servlet is ready to be loaded and initialized.

The Loading & Initialization Stage

In the loading and initialization stage of the JSP life cycle, the JSP container loads and instantiates the Servlet that has been generated and compiled in the translation and compilation stages, respectively. The Web container, as part of this process, performs the following operations:

- **Loading**—In this stage, the Web container loads the Servlet generated in the translation and compilation stages, using the normal Java class loading option. If the Web container fails to load the class, it terminates the loading process.
- **Instantiation**—After the Servlet class is loaded successfully, the JSP container creates an instance of the Servlet class. To create a new instance of the generated Servlet, the JSP container uses the no-argument constructor. The JSP translator (part of the JSP container) is responsible for including a no-argument constructor in the Servlet generated after the translation of the JSP.
- **Initialization**—After the JSP equivalent Servlet object is instantiated successfully, the JSP container initializes the instantiated object. As per the Servlet's life cycle, the container initializes the object by invoking the *init* (ServletConfig) method. This method is implemented by the container by calling the ***jspInit()*** method. This indicates that the *jspInit()* method acts as the *init(ServletConfig)* method of the Servlet.

If the Loading and Initialization stage is performed successfully, the Web container activates the Servlet object and makes it available and ready to handle client requests.

NOTE

The JSP page equivalent Servlet instance placed into service by a container may not handle any request in its lifetime.

The next stage in the JSP life cycle is request handling.

The Request Handling Stage

In the request handling stage of the JSP life cycle, the Web container uses only those objects of a JSP equivalent Servlet that are initialized successfully, to handle the client request. The container performs the following operations in this stage:

- Creates the ServletRequest and ServletResponse objects. If the client uses HTTP protocol to make the request, the container creates HttpServletRequest and HttpServletResponse objects, where the request object represents the request. That is, the request data and client information can be retrieved by the Servlet by using the request object. The response object can be used by the Servlet to generate the response.
- As per the Servlet's life cycle, the Web container invokes the *service ()* method on the Servlet object by passing the request and response objects created in the preceding step. In case of JSP equivalent Servlet, the container invokes the ***_jspService ()*** method.

Let's now look at the final stage of the JSP life cycle, that is, the destroying stage.

The Destroying Stage

If the Servlet container decides to destroy the JSP page's Servlet instance, that is, if it decides to end the services provided by the Servlet instance, it performs the following operations:

- It lets all the currently running threads in the service method of the Servlet instance complete their operations. Meanwhile, it makes the Servlet instance unavailable for new requests.
- After the current threads have completed their operations on the service() method, the container calls the *destroy()* method on a Servlet instance, which invokes the ***jspDestroy()*** method.
- After the destroy method is processed, the Servlet's life cycle is complete and the Web container releases all the references of the Servlet instance and renders them for garbage collection.

We now learn how to create simple JSP pages next.

Creating Simple JSP Pages

In this section, we are creating a simple Web application with two JSP pages. As we have not yet discussed the various constructs used in JSP, such as scriplets and directives, we create simple JSP pages to give you an idea of how they can be designed, used, and accessed by a Web application. Let's create the SimpleApp Web application containing two JSP pages. Create the Chapter 12 directory in the JavaEE directory (which you have created in *Chapter 11, Working with Servlet Programming*). Now, create the SimpleApp folder in the Chapter 12 directory. You can also find the SimpleApp application in the code\Java EE\Chapter 12\SimpleApp folder on the CD.

The first JSP page we will create is user.jsp, which provides an HTML form with two input fields to enter the name and city of a user. Listing 12.4 provides the code of the user.jsp page (you can find the user.jsp file in the code\Java EE\Chapter 12\SimpleApp folder):

Listing 12.4: The user.jsp File

```
<%@ page import="java.util.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
        <title>Welcome</title>
  </head>
  <body>
  <%=new Date().toString()%>
  <hr>
  <h2>Welcome</h2>
        <form action="showuser.jsp">
              <table>
                    <tr>
                          <td colspan="2" align="center">
                                Enter User Information
                          </td>
                    </tr>
                    <tr>
                          <td>User Name</td>
                          <td><input type="text" name="name"></td>
                    </tr>
                    <tr>
                          <td>City</td>
                          <td><input type="text" name="city"></td>
                    </tr>
                    <tr>
                    <td colspan="2"><input
                          type="submit" value="Enter"></td>
                    </tr>
              </table>
        </form>
  </body>
</html>
```

The second JSP page we create in this application is showuser.jsp, which shows the `name` and `city` that can be entered by the user through the input fields of the user.jsp page. Listing 12.5 shows the code of the showuser.jsp page (you can find the showuser.jsp file in the code\Java EE\Chapter 12\SimpleApp folder):

Listing 12.5: The showuser.jsp File

```
<%@ page import="java.util.*" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
        <title>Welcome</title>
  </head>
  <body>
  <%=new Date().toString()%>
  <hr>
  Hello <b><%=request.getParameter("name")%></b><br><br>
  Your City is <b><%=request.getParameter("city")%></b><br><br>
        <a href="user.jsp">Back</a>
  </body>
</html>
```

The directory structure of the Web application, which has been named SimpleApp, can be seen in Figure 12.3:

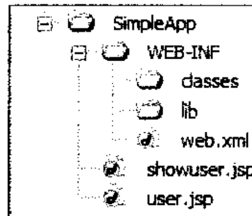

Figure 12.3: Showing Directory Structure of the SimpleApp Application

The WEB-INF\classes and WEB-INF\lib folders shown in Figure 12.3 are empty as we have not created any class or used any library, but are created here as per the standards for developing Web applications.

Since we are using JSPs, you do not need to provide any kind of mapping in the web.xml file. Listing 12.6 shows the code of the web.xml file: (you can find the web.xml file in the code\Java EE\Chapter 12\SimpleApp\WEB-INF folder on the CD):

Listing 12.6: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
</web-app>
```

You can simply place this SimpleApp project folder into the webapps folder located in the Tomcat installation directory to deploy it on the Tomcat server. Start your server and access the user.jsp page by entering the URL http://localhost:9090/SimpleApp/user.jsp into the Web browser. The output of the user.jsp page is shown in Figure 12.4:

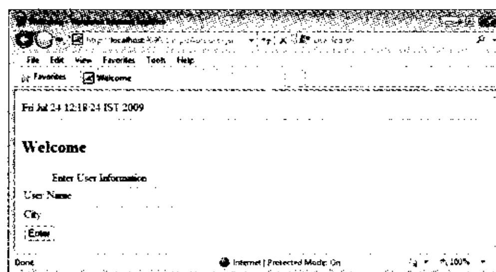

Figure 12.4: Showing Output of the user.jsp Page.

You can click the Enter button after entering a string into the User Name and City fields. Assuming that the user enters "Suchita" and "Delhi", respectively, you will see the showuser.jsp page, as shown in Figure 12.5:

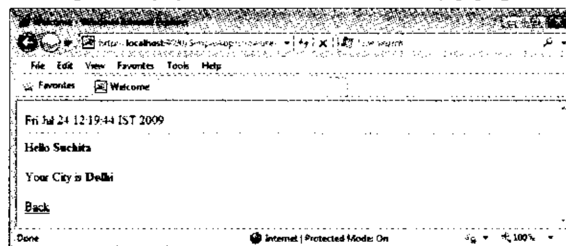

Figure 12.5: Showing Output of showuser.jsp page.

This was a simple application created with two JSP pages. Though the code written in Listing 12.4 and Listing 12.5 is quite simple and mostly contains HTML tags, other JSP constructs have also been used. For example, you can see JSP scriptlets enclosed in <%= %> and JSP directives enclosed in <%@ %>. Let's now discuss the JSP basic tags in detail.

Working with JSP Basic Tags and Implicit Objects

A JSP document contains elements or tags, such as directives, scripting elements, and implicit objects. Directives are used in a JSP document to include the content of a static or dynamic file within another file. Directives are also used to import Java files within JSP pages. Scripting elements, such as declaration tags, expression tags, and scriptlet tags, help declare variables and object references in a JSP page. Scripting elements also help generate and display dynamic content.

The Servlet API includes interfaces such as HttpServletRequest, HttpServletResponse, and HttpSession, which provide convenient abstractions to the developers. These abstractions encapsulate the object's implementation; for example, the HttpServletRequest interface represents the HTTP request sent from the client along with headers, form parameters, and so on, and provides convenient methods such as getParameter() and getHeader() that extract relevant data from the request. JSP implicit objects provide a convenient way to access the interfaces and objects of the Servlet API, without writing additional code.

NOTE

We have used both JSP page and JSP document to denote a JSP file. These terms have been used interchangeably. The JSP page is a term used to refer a JSP file written by using the traditional JSP syntax, whereas the term JSP document is used to refer a JSP file written by using XML syntax of JSP supported by the JSP 1.2 specification.

In this section, we learn about scripting and directive tags used in JSP. In addition, we learn to work with implicit objects.

Let's start by discussing JSP scripting tags.

Exploring Scripting Tags

JSP scripting tags, also called as JSP scripting elements, allow you to add a script code into the Java code of a JSP page generated by the JSP translator. In most of the cases, Java is the scripting language used to build a JSP page; however, other supported languages can also be used, depending on the Web containers.

You learn how to set the scripting language for a JSP page in the next section. However, before that, let's discuss about the different types of scripting tags.

Types of Scripting Tags

JSP defines three scripting tags, namely scriptlet, declarative, and expression tags. We describe these tags in detail next.

The Scriptlet Tag

Scriptlet tags allow you to write the script code (or the Java code) for implementing the_jspService method functionality. The JSP translator translates the statements of this tag into the _jspService method of the generated servlet. You can use a scriptlet tag to perform the following tasks:

- Declare variables that you can use later in a JSP page. That is, you can declare variables and write valid expressions in the page scripting language, within the scriptlet tags.
- Use any of the JSP implicit objects or any object declared with a <jsp:useBean> element.

Note that if the script code is not valid, an exception is thrown at the Compilation stage of the JSP lifecycle.

You can include multiple scriptlet tags in a single JSP document. The traditional JSP Syntax to use a scriptlet tag is as follows:

```
<%
   script code (allows multiple statements)
%>
```

The XML based syntax for the scriptlet tag is as follows:

```
<jsp:scriptlet>
   script code (allows multiple statements)
</jsp:scriptlet>
```

The following code snippet shows the implementation of the scriptlet tag:

```
<html><body>
<%
  for (int i=0;i<5;i++) {
        out.println(i);
  }
%>
</html></body>
```

In the preceding code snippet, the `for` loop of Java has been used in a JSP page to print numbers 1 to 5 in five lines. The servlet code for this JSP page, generated by the JSP container, is as follows:

```
public void _jspService(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
-
out.write("<html><body>");
for (int i=0;i<5;i++) {
      out.print(i);
}
out.write("</body></html>");
```

The Declarative Tag

The declarative tag allows you to write the script code that you need to provide in the generated Servlet class, outside the _jspService() method. This tag allows you to declare instance and class variables and implement class methods, such as jspInit and jspDestroy. The syntax of the declarative tag is as follows:

```
<%!
   script code (allows multiple statements)
%>
```

The XML based syntax for the declarative tag is as follows:

```
<jsp:declaration>
   script code (allows multiple statements)
</jsp:declaration>
```

The following code snippet shows the use of the declarative tag:

```
<%!
int a,b;
int fun(int a) {
        return a;
   }%>
<%a=1;%>
<%b=fun(a);%>
<%out.println(b);%
```

In the preceding code snippet, we declare two integer variables, a and b; and define a function, fun(), by using the declarative tag. Then, we assign 1 as the value to the `a` variable and pass it as an argument to the fun() function. The fun() function processes the argument and stores the result in the `b` variable. Finally, the value received by the `b` variable is displayed on the browser.

The Expression Tag

The expression tag allows you to write a Java expression, which is then resolved and the resultant value is displayed. The expression tag places the given Java expression in the out.print() method during the translation stage of the JSP life cycle. Consequently, the output of the Java expression is displayed with the output of the JSP page.

This tag also provides a simple and convenient way to access the script (i.e. Java) variables. The syntax of the expression tag is as follows:

```
<%=
  script code (allows only one expression)
%>
```

The XML based syntax for the expression tag is as follows:

```
<jsp:expression>
  script code (allows only one expression)
</jsp:expression>
```

The following code snippet shows the use of the expression tag:

```
<%! int count; %>
<html><body>
<% count++; %>
This page is requested by <%=count%> number of users till now.
</body></html>
```

The preceding code snippet can be embedded on a JSP page to count the number of requests received for the specific JSP page. Each time a new user accesses this Web page on the browser, the value of count is incremented by 1, which is displayed on the browser as well. For example, when a user accesses this page for the first time, the count variable is declared and is initialized at 0 (by default). The expression written in the scriptlet tag (<%count++%>) increments the value of the count variable by one and prints this value on the Web page. Now, when the other user accesses this page, the value of count is incremented by 1 and the browser display the result as 2.

You must consider the following points when using expression tags:

- Don't end an expression with ';' since the expression given in this tag is placed into the out.print() method.
- The expression given in this tag can be resolved into any type, such as int, float, double, boolean, String, or Object; but not to void.

If you write any code violating these rules, a translation stage error is raised. In addition, nested scripting tags are not allowed while using expression tags. It means a scripting tag cannot have another type of scripting tag. For example, in the following code snippet, the expression tag (<%= %>) is nested within the scriptlet tag (<% %>) which would raise a translation error:

```
<%
for (int i=0;i<5; i++){
<%=i%>   //out.println(+i);
}
%>
```

Working with JSP Scripting Tags

Let's now learn how to use JSP scripting tags, such as declaration, expression, and scriptlets in an application. Use of scripting tags varies as you go from building simple applications to complex ones. In simple applications, we use Servlet, JSP's scripting tags, with JavaBeans to build Web applications; whereas in complex applications, the scripting tags are used with custom tags and other Model-View-Controller (MVC) frameworks, such as Struts and JSF. In this subsection, we create the ScriptingElements application to demonstrate the use of the scripting tags.

The ScriptingTags.jsp page, as shown in Listing 12.7, uses page directives to import the java.util package and specify the scripting language as Java. In the declaration tag, we declare three integer variables: count, a, and b. We then define a function, fun(), which multiplies the integer value of 'a' variable, passed as an argument, with 10 and returns the result of the multiplication. The count variable is used to keep track of number of times this Web page is being accessed. Listing 12.7 shows the code for the ScriptingTags.jsp page (you can find the ScriptingTags.jsp file in the code\Java EE\Chapter 12\ScriptingElements folder on the CD):

Listing 12.7: The ScriptingTags.jsp File

```
<%@ page import ="java.util.*" language="java" %>
<html><body>
<center><h1>Using Scripting Elements</h1></center>
<%! int count,a,b;
int fun(int a) {
        return 10*a;
  }%>
  <% a=1;
  count++;
```

```
    for (int i=0;i<5;i++) {
        out.println("Value of i in iteration
        no."+i+":  <b>"+i+"</b><br/>");
    }
    b=fun(a);
    out.println("Value returned by fun():  <b>"+b+"</b><br/>");
%>
This page is requested by <b><%=count%></b> number of times on date
<b><%=new Date()%></b>.
</body></html>
```

You do not need to map this JSP page in the web.xml file; however, an empty web.xml file must exist at the appropriate place in the root directory. An empty web.xml configuration file is shown in the following code snippet:

```
<web-app/>
```

To run this application, perform the following steps:

1. Create a new Web Project and name it ScriptingElements.
2. Store the ScriptingTags.jsp and web.xml files in the ScriptingElements directory according to the standard directory structure of Web application. Figure 12.6 shows the directory structure of the application created in Listing 12.8:

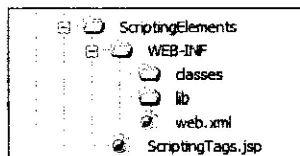

Figure 12.6: Showing Directory Structure of the ScriptingElements Application

3. Deploy the ScriptingElements folder on the Tomcat server.
4. Start the Tomcat server and browse the application by using the URL `http://localhost:9090/ScriptingElements/ScriptingTags.jsp`. The output of the application is displayed in the Web browser, as shown in Figure 12.7:

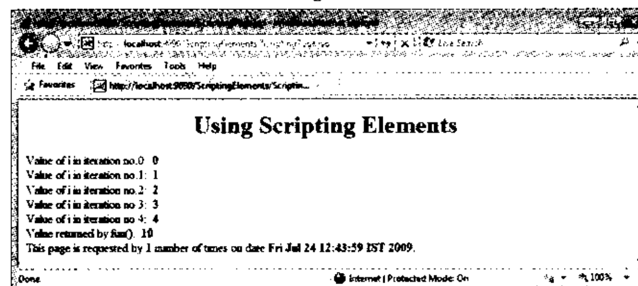

Figure 12.7: Showing the Use of Scripting Elements.

Figure 12.7 shows the use of JSP scripting tags in ScriptingTags JSP page. The scripting tags are used to iterate the values of the counter variables. Figure 12.7 also shows the number of times a user has requested the specific Web page.

After discussing about the scripting tags, let's now explain how to use the implicit objects in a JSP page.

Exploring Implicit Objects

JSP implicit objects are used in a JSP page to make the page dynamic. The dynamic content can be created and accessed by using Java objects within the scripting elements. JSP implicit objects are predefined objects that are accessible to all JSP pages. These objects are called implicit object because you don't need to instantiate these objects. The JSP container automatically instantiates these object while writing the script content in the scriptlet

and expression tags. JSP specification standardizes some object reference names, which are available for every JSP page so that any vendor-implemented translators have to take the responsibility of initializing these objects in the _jspService method.

Features of Implicit Objects

The following are the features of JSP implicit objects:

- JSP implicit objects allow developers to access services and resources provided by the Web container.
- JSP implicit objects are used to generate dynamic content of Web pages.
- JSP implicit objects are termed as implicit, so that you do not have to declare these object explicitly; the JSP container automatically instantiates these objects.
- JSP implicit objects are available to all JSP pages, and you can use these objects without importing them into your JSP page. Since the implicit objects are declared automatically, the user only needs to use a reference variable associated with a given object. This object can be used to call the methods associated with it.
- JSP implicit objects help in handling the HTML parameters, forward a request to a Web component, and include the content of a component, log data through the JSP container, control the output stream, and handle exceptions very easily.

Types of Implicit Objects

During the translation of a JSP page, the JSP engine initiates nine most commonly used JSP implicit objects in the _jspService() method. These JSP implicit objects are:

- Request
- Response
- Out
- Page
- PageContext
- Application
- Session
- Config
- Exception

Let's learn about these objects in detail.

The request Object

Request objects are passed as parameters to the _jspService() method when a client request is made. A request object is of the javax.servlet.http.HttpServletRequest type. You can use the request objects in a JSP page similar to using these in Servlets.

The response Object

The response object is used to carry the response of a client request after the _jspService() method is executed. The response object is of the javax.servlet.http.HttpServletResponse type.

The out Object

The out implicit object provides access to the servlet's output stream. This object is a subtype of javax.servlet.jsp.JspWriter. The out object can be used directly in JSP scriptlets to display the text data on the browser, since JSP expressions are automatically placed in the output stream. In other words, the out object is used to write the text data.

The page Object

The page object refers to the Servlet of the JSP page processing the current request, and is of the java.lang.Object type. It also works as an instance of the generated Servlet in a JSP page. Since it is a variable of type Object, it is rarely used in a document. It cannot be used directly to call the Servlet methods.

The pageContext Object

The pageContext object represents the context of the current JSP page. It provides a single API to manage the various scoped attributes. This object is of the javax.servlet.jsp.PageContext type.

The application Object

The application object refers to the entire environment of a Web application to which a JSP page belongs. This object is of the javax.servlet.ServletContext type.

The session Object

The session object helps access session data and is of the HttpSession type. It is declared if the value of the session attribute in a page directive is true. By default, the value of the session attribute is always true. If we explicitly specify the session attribute to false, the JSP engine does not declare this object. In such a case, if you try to access the session object, an error is generated.

The config Object

The config object specifies the configuration of the parameters passed to a JSP page. This object is of the javax.servlet.ServletConfig type. To demonstrate the use of the config object within a JSP page, we need to associate a servlet with a JSP file by using the <jsp-file> element. After associating a Servlet, all the initialization parameters for the named Servlet are made available to the JSP page by the ServletConfig object.

The exception Object

The exception object is of the java.lang.Throwable type and is only available for the JSP pages that act as the error handlers for other pages. This object is only available to the JSP error pages that have the isErrorPage attribute set to 'true' with page directive.

All objects discussed in this subsection are always available within the scriptlet and expression tags only. These objects are not applicable to use in declaration tag. However, use the getServletConfig() method if you want to get the ServletConfig type of object reference in a declaration tag while implementing the jspInit or jspDestroy methods. Use the getServletContext() method of ServletConfig to get the ServletContext object.

Working with Implicit Objects

Let's now learn how to use the implicit objects in a JSP page to create a dynamic Web page. Let's develop a simple Web application, ImplicitObjects, by using implicit objects, such as request, session, and pageContext. This application performs three tasks— overriding the jspInit() method to perform initializations, using initialization parameter to implement the config object, and accessing the JSP document placed in a private folder.

This application contains the following Web pages:

- Home.html—Represents the index page of the application
- request.jsp—Displays the welcome message
- pageContext.jsp—Demonstrates the use of the pageContext implicit object
- other.jsp—Demonstrates the use of other implicit objects such as page, session, out, application and config.

The Home.html page of this application consists of a simple form with a text field, Name, and a button, Invoke JSP. Listing 12.8 shows the code for the Home.html page (you can find the Home.html file in the code\Java EE\Chapter 12\ImplicitObjects folder on the CD):

Listing 12.8: The Home.html File

```
<html>
  <body>
        <form action="request.jsp">
                Name : <input type="text" name="name">
                <input type="submit" value="Invoke JSP"/>
        </form>
  </body>
</html>
```

When a user clicks the Invoke JSP button on the Home.html page, the next page, request.jsp page, is displayed. The request.jsp page displays a greeting message followed by the user name entered on the Home.html page. It then displays request details, such as type of request, its URI, and request protocol, in the tabular form. These request details are retrieved using the request implicit object's methods, such as getMethod(), getRequestURI( ), getProtocol( ), and getHeader( ). Listing 12.9 shows the code for the request.jsp page (you can find the request.jsp file in the code\Java EE\Chapter 12\ImplicitObjects folder on the CD):

Listing 12.9: The request.jsp File

```
<html>
  <head><title>
        Using Implicit Objects
        </title></head>
  <body>
  Hello, <b><%=request.getParameter("name")%></b><br/><br/>
  Your request details are <br/><br/>
  <table border="1">
  <tr><th>Name</th><th>Value</th></tr>
  <tr><td>request method</td>
  <td><%= request.getMethod() %></td></tr>
  <tr><td>request URI</td>
  <td><%= request.getRequestURI() %></td></tr>
  <tr><td>request protocol</td>
  <td><%= request.getProtocol() %></td></tr>
  <tr><td>browser</td>
  <td><%= request.getHeader("user-agent") %></td></tr>
  </table>
  <%if (session.getAttribute("sessionVar")==null) {
        session.setAttribute("sessionVar",new Integer(0));
  }%>
  <table>
        <tr><th align=left>Would you like to see use of remaining
                implicit objects?</th></tr>
        <tr>
        <form name=form1 action="pageContext.jsp" method="post">
        <td><input type="radio" name="other" value="Yes">Yes</td>
        <td><input type="radio" name="other" value="No" > No</td></tr>
        <tr><td><input type="submit" value="Submit"></td></tr>
        </form>
  </table>
  </body></html>
```

After submitting the form on the request.jsp page, the control is transferred to the pageContext.jsp page. The pageContext.jsp page uses the pageContext implicit object to forward a request to another JSP page, other.jsp. Listing 12.10 shows the pageContext.jsp page (you can find the pageContext.jsp file in the code\Java EE\Chapter 12\ImplicitObjects folder on the CD):

Listing 12.10: The pageContext.jsp File

```
<HTML>
  <HEAD>
  <TITLE> Intermediate </TITLE></HEAD>
  <BODY>
  <%
        if("Yes".equals(request.getParameter("other")))
        {
                pageContext.forward("other");
        }
  %>
  </BODY>
</HTML>
```

In the pageContext JSP page, the request is forwarded to the other JSP page. The other.jsp page shows the use of page, session, out, config, and application implicit objects. This page first overrides the jspInit function, which retrieves the initialization parameter of the other.jsp page from the web.xml file. The same initialization parameter value is fetched by calling the getInitParameter() method of the config implicit object (config.getInitParameter("count")). In the other.jsp page, we use the page implicit object's log method to log a message, anothermessage *(((HttpServlet)page).log("another rmessage");)* . The session implicit object is used to get the value of the *sessionVar* attribute, which shows the number of active sessions in the request.jsp page (.getAttribute("sessionVar")). The application implicit object is used to retrieve the value of the context parameter, param1, set in web.xml. We use the application.getInitParameter("param1") method to retrieve the value of the param1 parameter in this JSP page. Listing 12.11 shows the code for the other.jsp page (you can find the other.jsp file in the code\Java EE\Chapter 12\ImplicitObjects folder on the CD):

Listing 12.11: The other.jsp File

```
<html><body>
<%!int count;
   public void jspInit() {
   ServletConfig sc=getServletConfig();
   count=Integer.parseInt( sc.getInitParameter("count"));
   System.out.println("In jspInit");
   }
%>
Count value without using config implicit object: <b> <%=count%></b> <br/>
<%
this.log("log message");
((HttpServlet)page).log("anothermessage");
ServletContext ct = config.getServletContext();
out.println("Value of sessionVar
   is:"+"  <b>"+session.getAttribute("sessionVar")+"</b><br/>");
out.println("Server name and version using config implicit
   object:"+"  <b>"+ct.getServerInfo()+"</b><br/>");
out.println("Value of context parameter param1 get using application implicit
   object:"+"  <b>"+application.getInitParameter
         ("param1")+"</b><br/>");
out.println("Count value retrieved using config implicit
   object:" +"  <b>"+config.getInitParameter("count")+"</b>");
%>
</body> </html>
```

The web.xml file initializes an application level parameter, param1, with the value param1. In the ImplicitObjects Web application, we have configured the other.jsp page in web.xml, (although it is not mandatory to declare JSP page in web.xml, as described earlier). However, the other.jsp page has been declared in web.xml in our case due to the following reasons:

- Initialization parameter 'count' for the other.jsp page has been configured in web.xml.
- The other.jsp page has been placed into a private folder (jsppages) to avoid direct access to this JSP page.

The other.jsp page is configured using the <jsp-file> nested tag of the <servlet> tag. The servlet name specified in the <servlet-name> tag is 'myjsp', which is mapped to the URI for accessing this page.

Listing 12.12 shows the code for the web.xml file of the ImplicitObjects application (you can find the web.xml file in the code\Java EE\Chapter 12\ImplicitObjects\WEB-INF folder on the CD):

Listing 12.12: The web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <context-param>
         <param-name>param1</param-name>
```

```
<param-value>param1</param-value>
    </context-param>
    <servlet>
        <servlet-name>myjsp</servlet-name>
        <jsp-file>/other.jsp</jsp-file>
        <init-param>
            <param-name>count</param-name>
            <param-value>10</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>myjsp</servlet-name>
        <url-pattern>/other</url-pattern>
    </servlet-mapping>
</web-app>
```

To run this application effectively, you need to make a new Web Project, say ImplicitObjects. Place all files, such as Home.html, pageContext.jsp, web.xml and so on, under the directory structure of ImplicitObjects. The directory structure of the described Web application is shown in Figure 12.8:

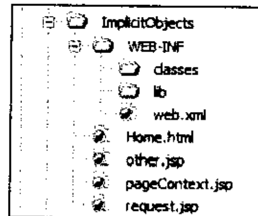

Figure 12.8: Showing Directory Structure of the ImplicitObjects Application

Now, deploy the ImplicitObjects folder on the Tomcat server. Browse the application using the URL `http://localhost:9090/ImplicitObjects/Home.html`. Figure 12.9 displays the home page of this application:

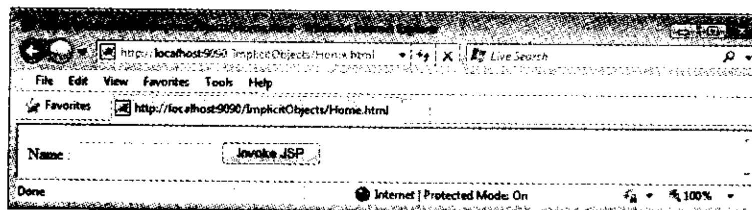

Figure 12.9: Displaying the Output of Home.html

When a user clicks the Invoke JSP button, the request.jsp page is displayed, as shown in Figure 12.10:

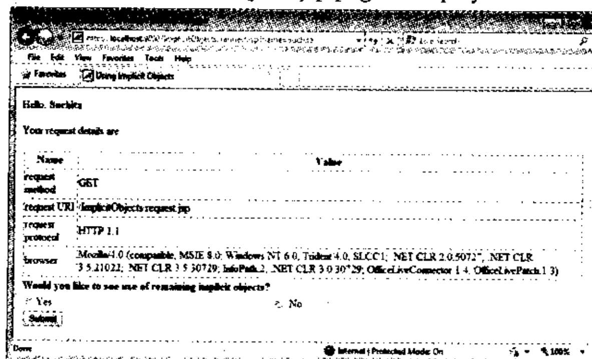

Figure 12.10: Displaying the Result of the request.jsp Page

Figure 12.10 shows all request details, such as request type, request URI, and request protocols, in the form of a table.

When a user selects the Yes radio button and clicks the Submit button, the other.jsp page is displayed, as shown in Figure 12.11:

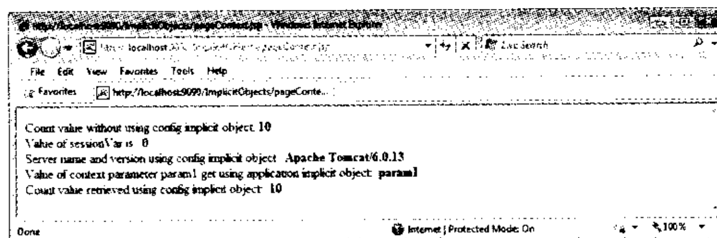

Figure 12.11: Displaying the Use of Implicit Objects.

Figure 12.11 shows the use of the implicit objects in an application. Figure 12.11 also shows the values of the implicit objects that are implicitly being used by the application. When the user selects any of the radio buttons shown in Figure 12.10 and clicks the Submit button, the pageContext.jsp page is invoked to display the values of the implicit objects used in the application.

Exploring Directive Tags

Directive tags are used to give directions used by the JSP translator during the translation stage of the JSP life-cycle. These tags are used to set global values, such as class declarations, methods to be implemented, and output content type.

Types of Directive Tags

According to the JSP specifications, three standard directive tags are available with all JSP compliant containers. These directive tags are:

- page
- include
- taglib

Let's discuss each of these directives in detail.

The Page Directive Tag

The page directive tag holds the instructions used by the translator during the translation stage of the JSP lifecycle. These instructions affect the various properties associated with the whole JSP page. You can use page directive multiple times in a JSP page. The page directive used on any part of the JSP page is automatically applied to the entire translation unit. However, it is often good programming style to place the page directive at the starting of the JSP page. The syntax of the page directive is as follows:

```
<%@page attributes %>
```

The XML based syntax for the page directive is as follows:

```
<jsp:directive.page attributes/>
```

There are total 11 attributes that can be used in the page directive tag. Table 12.1 lists these attributes:

Table 12.1: Attributes of page directive

Attribute	Value
language	Takes the scripting language as the value to be used in scripting tags. The default value is Java
import	Takes comma separated list of Java classes as the value
extends	Takes a complete qualified class name extended by the Servlet equivalent class written by the translator of the current JSP page

Table 12.1: Attributes of page directive

Attribute	Value
buffer	Takes the buffer size in kilobytes, for example none, 8kb, 16kb, 32kb, and 64kb. The default value is 8kb
autoFlush	Specifies whether the output has to be flushed automatically when the buffer is full. If this attribute is set to true, it automatically flushes the buffer as soon as the buffer is full. If this attribute is set to false, an exception is raised when buffer overflows. The default value of this attribute is true
isThreadSafe	Takes true or false as its value; the default value is true. This attribute specifies whether or not a JSP page is thread safe. That is, whether or not the instance of the servlet equivalent class of the JSP page is capable of handling simultaneous requests. If this attribute is set to false, only one thread can use the service provided by one object. If the value of this attribute is true, simultaneous requests can be handled by this page
errorPage	Takes the URL path of the page to which a request has to be dispatched when an exception is raised in the current page
isErrorPage	Takes true or false as its value; the default value is false. This attribute specifies whether or not the current page is an error page. Note that if this attribute is set to true, an additional implicit object, exception, is available for the current JSP page
contentType	Takes the response content MIME type, and optionally, character encoding. The default value is text/html
Session	Takes true or false, which indicates whether or not the session is required; the default value is *true*. If this attribute is set to *false*, the JSP page cannot use the session implicit object
Info	Takes a string, which can be retrieved by using the getServletInfo( ) method
pageEncoding	Specifies the encoding type to be used by the Web container to compile a JSP page. Some of the encoding types are ISO-8859-1 and UTF-8

The JSP page, as described earlier, can contain any number of page directive tags. However, except the import tag, none of the other tags are allowed to be specified more than once.

The Include Directive Tag

The include directive tag is used to merge the contents of two or more files during the translation stage of the JSP lifecycle. The include directive adds the text of the included file to the JSP page, without any processing or modification. The included file can be static or dynamic. If the included file is dynamic, its JSP elements are translated and included. In this case, if there are any changes in the included page after the included JSP page is translated, the changes are not applied until the included JSP page is translated once again.

The syntax of the include directive is as follows:

```
<%@include file=" file path" %>
```

The XML based syntax for include directive is as follows:

```
<jsp:directive.include file=" file path "/>
```

The following code snippet shows the implementation of the include directive for merging the Test.jsp and Test1.html files:

```
<%@include file="/Test.jsp" %>
<%@include file="/Test1.html" %>
```

Note that the behavior of the include directive with respect to the recompilation of the included page, if any changes are done to it, depends on the Web container. This means that if the included page is changed, some containers recognize and apply the changes to the JSP page, while others do not.

The Taglib Directive Tag

The taglib directive tag is used to declare a custom tag library in a JSP page so that the tags related to that custom tag library can be used in the same JSP page. The syntax of taglib directive is as follows:

```
<%@taglib uri=" URI" prefix=" unique_prefix " %>
```

We do not have any special XML equivalent element for this tag since this declaration is done by using the XML namespace syntax.

Working with JSP Directive Tags

JSP directives are used in a JSP page to add functionality to the JSP page. Let's create the directiveTags application using the JSP directives. This application contains the following pages:

- **Login.html** – Accepts user details and redirects the user to the required JSP page.
- **LoginProcess.jsp** – Connects to a database to process the user request.
- **MyError.jsp** – Represents the JSP page that is called when an error or exception is raised.

In this application, we use the MySQL 5.0 database to access the related tables in the application. Therefore, you need to create the database table to work with the application before creating these JSP pages. The code to create the table is as follows:

```
create database sample;
use sample;
create table userdetails(uname varchar(15),pass varchar(10));
insert into userdetails values('Suchita','123456');
```

Let's now create these pages.

Listing 12.13 shows the Login.html page, which is the home page of the application (you can find the Login.html file in the code\Java EE\Chapter 12\directiveTags folder on the CD):

Listing 12.13: The Login.html File

```
<html> <body><pre>
  <form action="LoginProcess.jsp">
        <b>User Name</b> : <input type="text" name="uname"/>
        <b>Password</b>  : <input type="password" name="pass"/>
        <input type="submit" value="LogIN"/>
  </form> </pre></body></html>
```

The Login.html page allows the user to enter a user name and password, which are stored in a database.

After entering the user name and password, the LoginProcess.jsp page is called to handle the request. Listing 12.14 shows the code for the LoginProcess.jsp page (you can find the LoginProcess.jsp file in the code\Java EE\Chapter 12\directiveTags folder on the CD):

Listing 12.14: The LoginProcess.jsp File

```
<%@page import="java.sql.*" errorPage="/MyError.jsp"%>

<html><body>

<%
  Connection con=null;
  String uname=request.getParameter("uname");
  String pass=request.getParameter("pass");
  try {
        Class.forName("com.mysql.jdbc.Driver");
        con=DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/sample","root","root");
        Statement st=con.createStatement();
        ResultSet rs=st.executeQuery(
        "select * from userdetails where uname=\'"+uname+ "\'
              and pass=\'"+ pass +"\'");
        if (!rs.next()){
%>
```

```
User details given for uname : <%=request.getParameter("uname")%> are not
valid <br/> Try again
<%@include file="Login.html"%>
</body></html>
<%
  return;
 }//if
 }//try
 finally {
      try{con.close();}catch(Exception e){}
 }//finally
%>
This is a Home Page <br/>
Welcome, <%=uname%>

</body>

</html>
```

The LoginProcess.jsp page shows the use of the JSP page directive, which imports the java.sql package into the LoginProcess JSP page. The java.sql package helps the LoginProcess.jsp JSP page to access the database that stores user information. After the request is submitted by the user, the LoginProcess.jsp page is used to generate the appropriate response for each request made.

The LoginProcess.jsp page also includes another page, named MyError.jsp, which is used to handle any exception that may occur in the JSP page. Any exception in the LoginProcess.jsp page is forwarded to the MyError.jsp. Listing 12.15 shows the code for the MyError.jsp page (you can find the MyError.jsp file in the code\Java EE\Chapter 12\directiveTags folder on the CD):

Listing 12.15: The MyError.jsp File

```
<%@page isErrorPage="true"%>
<html><body>
  An exception is raised while processing the request: <br/>
  <b>Exception : </b> <br/>
      <%=exception.getMessage()%>
</body> </html>
```

The files used in this application need to be placed properly in a separate directory named directiveTags. Since the application imports the java.sql package, you need to include the mysql-connector-java-5.0.4-bin.jar files in the lib directory of the application. This is an executable jar file used to access a database within a JSP page. In this case, we have included mysql-connector-java-5.0.4-bin.jar file in the directory since we are using the MySQL 5.0 database for the application. Figure 12.12 shows the directory structure of the described application:

```
directiveTags
  WEB-INF
    classes
    lib
      mysql-connector-java-5.0.4-bin.jar
  Login.html
  LoginProcess.jsp
  MyError.jsp
  web.xml
```

Figure 12.12: Directory Structure for the directiveTags Application

The application needs to be deployed on the Web server to be accessed by the users. Copy the directiveTags folder into the webapps folder of the Tomcat installation.

After placing the files as per the directory structure described above, you can access the application by using the URL `http://localhost:9090/directiveTags/Login.html`. Figure 12.13 shows the Login.html page as it appears on the Web browser:

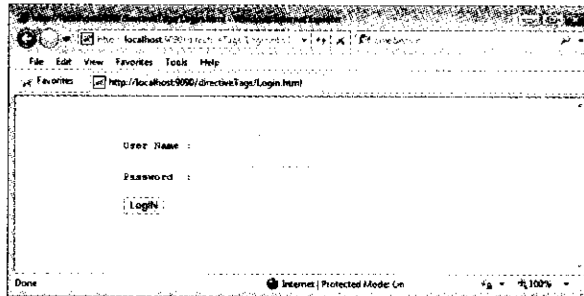

Figure 12.13: Displaying the Output of Login.html

Enter the username and password on the Login.html page. After entering the user name and password, the JSP engine checks for the availability of the user name and the password in the specified database. If the entries are valid, you are directed to the LoginProcess.jsp page, as shown in Figure 12.14:

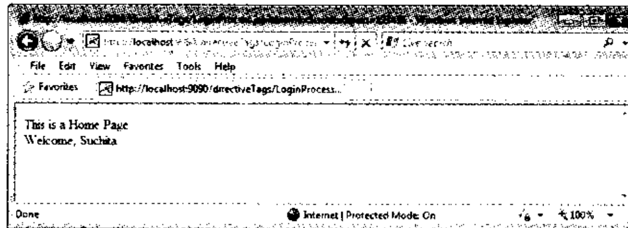

Figure 12.14: Displaying the Output of LoginProcess.jsp

Figure 12.14 shows the output for the successful login in the Login.html page. The database is called each time a user enters the values for the user name and password fields. If the entries are not available in the database, the user is redirected to another page, as shown in Figure 12.15:

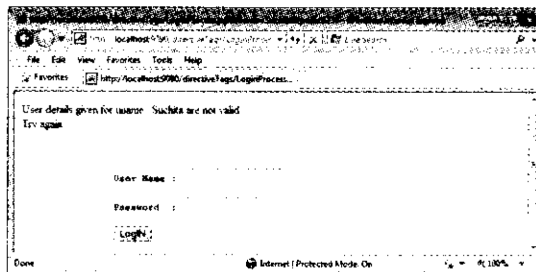

Figure 12.15: Displaying Output of LoginProcess.jsp for an Invalid Login

Now, to test the errorPage and isErrorPage functionalities, drop the table from the database, by using the following command:

```
drop table userdetails;
```

If you make a request to the LoginProcess.jsp by clicking the **Login** button, a SQLException is raised in the LoginProcess page.

After discussing the JSP basic tags and implicit objects, let's now discuss how to use JavaBeans and action tags in JSP.

Using JavaBeans and Action Tags in JSP

Use of JavaBeans with JSP has made the work of developers easy because developers do not need to handle long Java syntax within a JSP page. We start this chapter by introducing JavaBeans. In addition, we try to analyze why JavaBeans are required when we can work with JSP. We start the discussion by describing the advantages

of using JavaBeans. The components and properties of JavaBeans are described next. We also examine how JavaBeans can be used with JSP. Finally, we learn about JSP Action tags. Let's start discussing about JavaBean.

What is JavaBean?

JavaBeans are reusable software components that separate the business logic from the presentation logic. In general, JavaBeans are simple Java classes that follow certain specifications to develop dynamic content. JavaBeans are required to create effective and dynamic Web pages by providing the benefit of using separate Java classes instead of embedding large amount of Java code directly in a JSP page. These separate JavaBean classes are easier to write, compile, test, debug, and reuse. JavaBean uses getter and setter methods to invoke various methods explicitly to use their functionality with JSP pages.

You can create reusable and platform-independent JavaBean components with the JavaBeans API. JavaBean components are also known as beans. JavaBeans can be created or manipulated by using JavaBeans-compliant application builder tools such as NetBeans IDE. The JavaBeans-compliant application builder tools automatically discover information about the classes, based on JavaBeans specifications, and can create and manipulate these classes.

Advantages of using JavaBeans

JavaBeans have many advantages over scriptlets and JSP expressions. Some advantages of using JavaBeans are as follows:

- With the use of JavaBeans, users can manage presentation code and business logic separately. This feature of JavaBeans is more advantageous in big organizations that have separate Web and Java development teams. Both the presentation logic and business logic are developed separately, and JavaBeans ensure proper communication between them.
- Using JavaBeans with JSP has made object sharing between multiple Web pages simple.
- Using JavaBeans with JSP has simplified the process of request and response handling.

These are the advantages of using JavaBeans. Now, we learn about using the JavaBeans component with JSP action tags.

NOTE

A JavaBean component refers to any Java class that follows certain design conventions.

Action Tags

Action tags have been introduced in JSP 1.1, and some additional tags have been added in the JSP 1.2 and 2.0 specifications. These set of tags allow us to include some basic actions, such as inserting some other page resources, forwarding the request to another page, creating or locating the JavaBean instances, and setting and retrieving bean properties, in JSP pages.

The action tags are specific to a JSP page. When the JSP container encounters an action tag while converting a JSP page into a servlet, it generates the Java code that corresponds to the required predefined task. For example when it comes across the 'include' action tag:

```
<jsp:include page="myjsp.jsp" flush="true" />
```

In the preceding code snippet, the jsp:include action tag allows to include the myjsp.jsp page.

Some important action tags available in JSP are as follows:

- `<jsp:include>`
- `<jsp:forward>`
- `<jsp:param>`
- `<jsp:useBean>`
- `<jsp:setProperty>`
- `<jsp:getProperty>`
- `<jsp:plugin>`

- <jsp:params>
- <jsp:fallback>
- <jsp:attribute>
- <jsp:body>
- <jsp:element>
- <jsp:text>

Let's now describe these action tags in detail.

Describing the Include Tag

The include action tag allows a static or dynamic resource such as HTML or JSP page, specified by a URL, to be included in the current JSP while processing a request. If the included resource is static, its content is included in the calling JSP page. If the resource is dynamic, it acts on a request and sends back a result, which is included in the JSP page. For example, if the resource is another JSP page, the output content generated by the second JSP page is included in the first JSP page. The following code snippet shows the syntax of the include action tag in a JSP page:

```
<jsp:include attributes>
<!-- Zero or more jsp:param tags -->
</jsp:include>
```

NOTE

If the included resource is dynamic, we can use the jsp:param tag to pass the name and value of a parameter to the resource, as you can see in the previous code snippet. You learn about jsp:param later in this chapter.

The include action tag has certain attributes, which are used to accept values for the specified JSP page. The attributes specific to the include tag are as follows:

- **page**—Takes a relative URL, which locates the resource to be included in the JSP page. This allows us to give an expression that evaluates to a String equivalent to the relative URL that locates the resource. We can specify the relative URL directly or in an expression within a page attribute. The following syntax shows the use of the page attribute in a JSP page:
  - <jsp:include page="/Header.html"/>
  - <jsp:include page="<%=mypath%>"/>

  The relative URL cannot contain a protocol name, port number, or domain name. The relative URL starts with the '/' (forward slash) character, specifying that the URL is taken relative to the context path.
- **flush**—Takes true or false, which indicates whether or not the buffer needs to be flushed before including the resource. If the value is true, the buffer is flushed before including the resource. The default value of this attribute is false. After the include action is completed, the JSP container continues to process the rest of the JSP page.

You might be thinking how is the include tag different from the include directive tag described in the *Working with JSP Basic Tags and Implicit Objects* section. Now, let's discuss the difference between the 'include' directive tag and the 'include' action tag.

Difference Between Include Directive and Include Action Tags

The 'include' directive tag inserts the given page and includes the content in the generated Servlet page during the translation phase of the JSP lifecycle. In general, the 'include' directive is used to include files, such as HTML, JSP, XML, or a simple .txt file, into a JSP page statically. The file attribute is the only mandatory attribute available in the include directive tag and is used to refer to the file to be included. The following code snippet shows the use of the JSP include directive tag in a JSP file:

```
%@include file="menu.jsp"
```

In the preceding code snippet, menu.jsp is the file included in the current JSP page. The content of the menu.jsp page is included in the current JSP page during the translation stage. The include directive tag does not allow you to use the expression for giving the URL path, for including the resource in the JSP page.

The 'include' action tag is used to include the response generated by executing the specified JSP page or Servlet. The response is included during the request processing phase, when the page is requested by the user. The specified JSP page is included in request handling phase of the included JSP page lifecycle. The following code snippet shows the use of the include action tag in a JSP file:

```
<jsp:include page= "menu.jsp">
```

In the preceding code snippet, the output of menu.jsp needs to be included in the current page. The respective logic for include is written into the Servlet equivalent generated for the current JSP page.

Unlike the include directive tag, the include action tag accepts expressions. That is, with the include action tag, we can decide the page to be included at runtime, whereas it is not possible with the include directive tag.

Describing the Forward Tag

The jsp:forward tag forwards a JSP request to another resource, which can be either static or dynamic. If the resource is dynamic, we can use a jsp:param tag to pass the name and value of the parameter to the resource. We use the page directive with the buffer="none" parameter to specify that the output of the JSP page should not be buffered.

If the output stream is not buffered, and some output has been written to it, a <jsp:forward> action throws a java.IllegalStateException. The forward action in a JSP page works similar to the `forward()` method of RequestDisapatcher. The following code snippet shows the use of the jsp:forward action in a JSP page:

```
<jsp:forward attributes>
<!-- Zero or more jsp:param tags -->
</jsp:forward>
```

The jsp:forward action takes a page attribute to specify the page to which the current request is to be forwarded. The page attribute takes the relative URL that locates the resource to which the request has to be forwarded. This allows us to give an expression that evaluates to a String equivalent to the relative URL that locates the resource. We can specify the relative URL directly or by using an expression. The following code snippet shows the use of the page attribute in the jsp:forward action:

```
<jsp:forward page="/Header.html"/>
<jsp:forward page="<%=mypath%>"/>
```

The given URL cannot contain a protocol name, port number, or domain name. If the given URL starts with the / character, the URL is taken relative to the context path, if not relative to the JSP page. After the forward action is completed, the JSP container does not continue processing the rest of the JSP page.

Describing the Param Tag

The jsp:param tag allows us to pass a name and value pair as parameter to a dynamic resource, while including it in a JSP page or forwarding a request from a JSP page to another JSP page. The <jsp:param> tag can be even used to pass parameters to an applet configured in JSP page by using the jsp:plugin tag. The following syntax shows the use of the param tag in a JSP page:

```
<jsp:param attributes />
```

We can use more than one jsp:param tags if we want to pass more than one parameter. The param action tag contains two attributes, name and value, which are discussed next.

The name Attribute

The name attribute specifies the parameter name and takes a case-sensitive literal string. The parameter name should be unique, that is, it should be different for each parameter.

The value Attribute

The value attribute specifies the parameter value and takes either a case-sensitive literal String or an expression that is evaluated in the request handling stage of the JSP lifecycle. The following snippet shows the use of the JSP param and include actions, along with the attributes of jsp:param action:

```
<jsp:include page="/MyPage2.jsp">
<jsp:param name="uname" value="User1"/>
<jsp:param name="user_type" value="admin"/>
</jsp:include>
```

Let's create the actiontags_ex1 application to demonstrate the functionality of the discussed action tags in a JSP page. The Home.html page is used to create the user interface where the user needs to enter the values for the specified fields. Listing 12.16 shows the Home.html page (you can find the Home.html file in the code\Java EE\Chapter 12\actiontags_ex1 folder on the CD):

Listing 12.16: The Home.html File

```
<html> <body> <pre>
<form action="FrontJSP.jsp">
  <b>Operand 1 :</b> <input type="text" name="field1"/>
  <b>Operand 2 :</b> <input type="text" name="field2"/>
  <input type="submit" name="submit" value="Add"/>
  <input type="submit" name="submit" value="Sub"/>
</form> </pre> </body></html>
```

When you execute Listing 12.16, the Home.html page is displayed, which provides options to enter the values of certain fields. After entering the values for the required fields in the Home.html page, the control is transferred to the FrontJsp.jsp. Let's now create the FrontJsp.jsp page, as shown in Listing 12.17 (you can find the FrontJSP.jsp file in the code\Java EE\Chapter 12\actiontags_ex1 folder on the CD):

Listing 12.17: The FrontJSP.jsp File

```
<%@page errorPage="/Home.html"%>
<%
String s1=request.getParameter("field1");
String s2=request.getParameter("field2");
Integer.parseInt(s1); Integer.parseInt(s2);
String submit=request.getParameter("submit");
if (submit.equals("Add")) {
%>
  <jsp:forward page="/AddJsp.jsp"/>
<%
} else if (submit.equals("Sub")){
%>
  <jsp:forward page="/SubJsp.jsp"/>
<%
} else {
%>
  <jsp:forward page="/Home.html"/>
<%}%>
```

The FrontJSP.jsp page provides two buttons, Add and Subtract. From the FrontJSP.jsp, the control can be transferred to the AddJsp.jsp or SubJsp.jsp page, depending on the button that a user clicks. The jsp:forward action is used to forward the action to the respective pages. Listing 12.18 shows the code for AddJsp.jsp (you can find the AddJsp.jsp file in the code\Java EE\Chapter 12\actiontags_ex1 folder on the CD):

Listing 12.18: The AddJsp.jsp File

```
<%
  int f1=Integer.parseInt(request.getParameter("field1"));
  int f2=Integer.parseInt(request.getParameter("field2"));
  int result=f1+f2;
%>
<jsp:forward page="/Result.jsp">
<jsp:param name="result" value="<%=result%>"/> </jsp:forward>
```

The AddJsp.jsp is called if the user clicks the Add button. The AddJsp.jsp page is used to add the values entered by the user. This page then transfers the control to the Result.jsp page to view the result of addition.

Similarly, if the user clicks the Sub button, the SubJsp.jsp page is displayed. Listing 12.19 shows the code for the SubJsp.jsp page of the application (you can find the SubJsp.jsp file in the code\Java EE\Chapter 12\actiontags_ex1 folder on the CD):

Listing 12.19: SubJsp.jsp

```
<%
   int f1=Integer.parseInt(request.getParameter("field1"));
   int f2=Integer.parseInt(request.getParameter("field2"));
   int result=f1-f2;
%>
<jsp:forward page="/Result.jsp">
   <jsp:param name="result" value="<%=result%>"/>
</jsp:forward>
```

The SubJsp.jsp page also transfers the control to the Result.jsp page. Listing 12.20 shows the Result.jsp page (you can find the Result.jsp file in the code\Java EE\Chapter 12\actiontags_ex1 folder on the CD):

Listing 12.20: The Result.jsp File

```
<%
   String result=request.getParameter("result");
   String submit=request.getParameter("submit");
%>
<html><body><center>
   <%
        if (submit.equals("Add")){
   %>
   Result of Add : <%=result%>
   <%
        } else {
   %>
   Result of Sub : <%=result%>
   <%}%>
   </center>
   <jsp:include page="/Home.html"/>
</body></html>
```

The pages designed above must be mapped in the web.xml file to access them in the browser. Listing 12.21 shows the web.xml file associated with the preceding application (you can find the web.xml file in the code\Java EE\Chapter 12\actiontags_ex1\WEB-INF folder on the CD):

Listing 12.21: The web.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <welcome-files-list>
        <welcome-file>Home.html</welcome-file>
   </welcome-files-list>
</web-app>
```

All the specified files are to be placed under a common directory named actiontags_ex1. Place this directory in the <tomcat-home>/webapps directory and start the Tomcat server. Figure 12.16 shows the directory structure of the actiontags_ex1 folder:

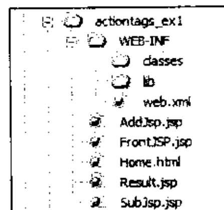

Figure 12.16: Showing Directory Structure for the actiontags_ex1 Application

Now, browse the application using the URL, `http://localhost:9090/actiontags_ex1/Home.html`, as shown in Figure 12.17. From this view, you can add or subtract any two numbers. Enter some numbers, as shown in Figure 12.17:

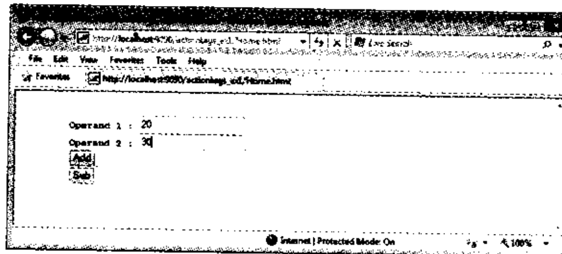

Figure 12.17: Showing Output of the Home.html Page

If the user enters 20 and 30 as the values for the Operand 1 and Operand 2 fields respectively, (Figure 12.17) and clicks the Add button, the control is transferred to the AddJsp.jsp page. Figure 12.18 shows the AddJsp.jsp page, as it appears after the user clicks the Add button:

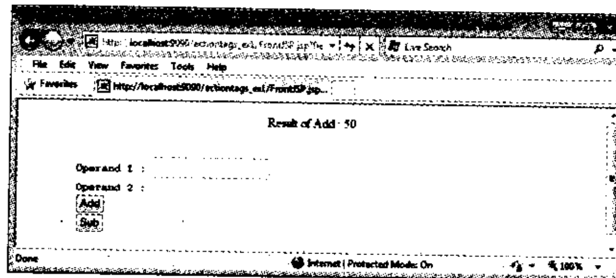

Figure 12.18: Displaying Result.jsp forwarded from AddJsp.jsp.

Figure 12.18 shows the result of clicking the Add button on the Home.html page.

Similarly, if the user clicks the Sub button on the Home.html page after entering 60 and 30 as the values of Operand 1 and Operand 2, the control is transferred to the SubJsp.jsp page. Figure 12.19 shows the result of clicking the Sub button:

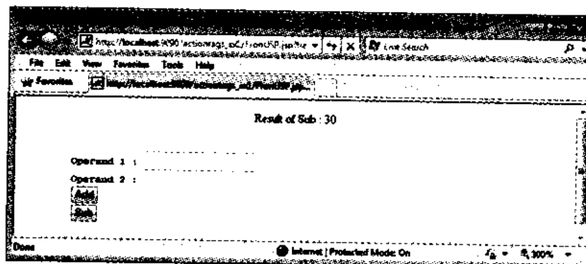

Figure 12.19: Displaying Result.jsp forwarded from SubJsp.jsp.

Figure 12.19 shows the Result.jsp page when the user clicks the Sub button.

Describing the useBean Tag

To separate logic from presentation, it is often a good idea to encapsulate logic in a Java object (a JavaBean), and then instantiate and use this object within a JSP page. The <jsp:useBean>, <jsp:setProperty>, and <jsp:getProperty> tags assist with this task.

The <jsp:useBean> action tag is used to instantiate a JavaBean, or to locate an existing bean instance, and assign it to a variable name (or `id`). We can also specify the lifetime of the object by giving it a specific `scope`, such as application scope. The <jsp:useBean> action tag ensures that the object is available, with the specified `id`, in the appropriate `scope` as specified by the tag. The object can then be referenced using its associated id from within

the JSP page, or even from within other JSP pages, depending upon the scope of the JavaBean. The syntax of the <jsp:useBean> action tag is as follows:

```
<jsp:useBean attributes>
<!-optional body content-->
</jsp:useBean>
```

Describing the Attributes of the <jsp:useBean> Tag

The attributes add extra characteristics to a tag, irrespective of its basic functionality. For example, the scope attribute of the jsp:useBean action tag makes a Java bean instance available in different scopes. Some attributes specific to the useBean tag are:

- **id**—Represents the variable name assigned to the id attribute of the jsp:useBean action tag. The id attribute is used to locate an existing bean instance in the appropriate scope specified in the jsp:useBean action tag. The id attribute is case sensitive and must follow the naming conventions used to define Java variables.
- **Scope**—Represents the scope in which the bean instance has to be located or created. Scopes can be page, request, session, and application. The default scope is page. All the scopes are defined as follows:
  - **page scope**—Indicates that the bean can be used where the <jsp:useBean> action tag is used within the JSP page, until the page sends a response back to the client or forwards a request to another resource.
  - **request scope**—Indicates that the bean can be used from any JSP page that is processing the same request, until a JSP page sends a response to the client.
  - **session scope**—Indicates that the bean can be used from any JSP page invoked in the same session as the JSP page that created the bean. The bean exists throughout the entire session, and can be accessed by any page that shares the session. Note that the page in which you create the bean must have a page directive with session="true".
  - **application Scope**—Indicates that the bean can be used from any JSP page in the same application as the JSP page that created the bean. The bean exists throughout an entire Web application, and can be accessed by any page in the application.
- **class**—Accepts the qualified class name to create a bean instance if the bean instance is not found in the given scope. The class name given in this attribute should have a no-argument constructor and should not be an abstract class. This is because the JSP container uses the *new* keyword and no-argument constructor to create a bean instance.
- **beanName**—Accepts a qualified class name or an expression that resolves to a qualified class name or serialized template. The JSP container uses instantiate method of the java.beans.Beans class to instantiate the bean. The instantiate method of the java.beans.Beans class instantiates the given bean in the following approach:
  - First it checks whether the given name represents the serialized template, if found, then it reads the serialized template by using the class loader to instantiates the bean, the serialized form will be located in the file with a name packagename.classname.ser.
  - If the given name does not represent a serialized template, then it creates an instance by using the no-argument constructor.
- **type**—Accepts a qualified class or interface name, which can be the class name given in the class or beanName attribute or its super type. This attribute can be used with or without class or beanName. If this attribute is used with the class or beanName attribute, the bean located or created will be taken into the reference variable of the type as given under this attribute. The variable name will be as specified in the id attribute. If this attribute is used without the class or beanName attribute, the JSP container only tries to locate the Bean but does not instantiate the Bean. If the Bean is not found, the JSP container throws InstantiationException.

Using the <jsp:useBean> Tag Attributes

The <jsp:useBean> tag attributes can be used in different combinations, as shown in Listing 12.22:

Listing 12.22: Various ways of using <jsp:useBean> tag

```
<jsp:useBean id="mybean" class="com.kogent.jspex.MyBean" scope="session"/>
<jsp:useBean id="mybean" beanName="com.kogent.jspex.MyBean"/>
<jsp:useBean id="mybean" class="com.kogent.jspex.MyBean" scope="session"
  type="com.kogent.jspex.MyBeanIntf"/>
<jsp:useBean id="mybean" beanName="com.kogent.jspex.MyBean" scope="session"
  type="com.kogent.jspex.MyBeanIntf"/>
<jsp:useBean id="mybean" scope="session" type="com.kogent.jspex.MyBeanIntf"/>
```

If you use the <jsp:useBean> action tag in a JSP page, you must include the logic into the Java code generated for the JSP page. The generated logic performs the following operations:

1. Declares an object reference variable with the given name (that is, the value of the id attribute) of the type specified in the class or beanName attribute. If the type attribute is already specified, the reference variable should be of the type specified in the type attribute.
2. Determines whether or not the bean instance is available in the given scope with the given name. If the bean is found in the given scope, the reference of the bean instance is stored in the reference variable, as described in previous step.
3. If the bean cannot be located in the given scope, the <jsp:useBean> action tag instantiates the class specified in the class or beanName attribute and then stores a reference of that class in the new reference variable. Thereafter, the <jsp:useBean> action tag executes its body content if it has body content.
4. If the bean cannot be located in the given scope, the class or beanName attribute is not assigned to the class or bean reference, and only type attribute is used, an InstantiationException is thrown.

Generating Servlets with the <jsp:useBean> Tag

Let's try to evaluate how a Servlet is created after a JSP page using some combinations of the <jsp:useBean> tag is compiled. For simplicity, we take some combinations of the <jsp:useBean> tag specified in Listing 12.23. Suppose your JSP page contains the following <jsp:useBean> tag:

```
<jsp:useBean id="regDetails" class="com.kogent.jspex.UserDetails" scope="session"/>
```

Let's look at how this <jsp:useBean> tag is processed by the Web container. Listing 12.23 shows the Servlet code generated by the Tomcat Web container:

Listing 12.23: Generated Servlet code

```
public void_jspService(
HttpServletRequest request,
HttpServletResponse response)
   throws java.io.IOException, ServletException {
   com.kogent.jspex.UserDetails regDetails = null;
         synchronized (session) {
         regDetails = (com.kogent.jspex.UserDetails)
         _jspx_page_context.getAttribute("regDetails",
         PageContext.SESSION_SCOPE);
         if (regDetails == null){
                regDetails = new com.kogent.jspex.UserDetails();
                _jspx_page_context.setAttribute("regDetails", regDetails,
                PageContext.SESSION_SCOPE);
   }//if
   }//synchronized
```

The implementation logic to create the Servlet may be different for other servers since most of the servers use built-in utility classes to perform these operations.

Let's look at another example next. Now suppose that your JSP page contains the following <jsp:useBean> action tag:

```
<jsp:useBean id="regDetails" type="com.kogent.jspex.UserDetails" scope="session"/
```

Notice that Listing 12.23 generates an exception if the specified instance type is not found. In our second example, the type attribute is specified with the <jsp:useBean> tag without using the class attribute. Listing 12.24 shows the Servlet code generated with the second example of the <jsp:useBean> tag:

Listing 12.24: Generated Servlet Code

```
public void _jspService(
HttpServletRequest request,
HttpServletResponse response)
throws java.io.IOException, ServletException {
...
  com.kogent.jspex.UserDetails regDetails = null;
  synchronized (session) {
        regDetails = (com.kogent.jspex.UserDetails)
        _jspx_page_context.getAttribute("regDetails",
        PageContext.SESSION_SCOPE);
        if (regDetails == null){
              throw new java.lang.InstantiationException(
              "bean regDetails not found within scope");
        }//if
  }//synchronized
...
}
```

In Listing 12.23, if the specified class is not found, UserDetails, a new instance of the UserDetails class, is created and bound to the specified scope. In Listing 12.24, the type attribute is specified and no class attribute has been provided, so if 'regDetails' does not exist in the "session" scope, then Instantiation Exception is thrown.

In this topic, we learned about using the <jsp:useBean> tag. Let's now learn about the <jsp:setProperty> tag next.

Describing the setProperty Tag

The <jsp:setProperty> action tag sets the value of a property in a bean, using the bean's setter methods. Before using the < jsp:setProperty> action tag; the bean must be instantiated. In addition, the name attribute of the bean must be the same as the reference variable name of the bean instance. The bean can be instantiated by using the <jsp:useBean> action tag as explained earlier. In this case, the attribute name of the <jsp:setProperty> action tag must be the same as the attribute id of <jsp:useBean>. Since both tags work together, these attributes are used to ensure co-ordination between them. The syntax of the < jsp:setProperty> tag is as follows:

```
<jsp:setProperty attributes/>
```

The <jsp:setProperty> tag contains various attributes, such as:

- name
- property
- value
- param

Let us now discuss these attributes in detail.

The name Attribute

The name attribute takes the name of the already existing bean as a reference variable to invoke the setter method. This attribute must match with the value of the id attribute of <jsp:useBean> to set the properties of a bean. Consequently, the <jsp:useBean> tag must be declared before the <jsp:setProperty> tag in a JSP page.

The property Attribute

The property attribute of the <jsp:setProperty> tag takes property name that has to be set, and specifies the setter method that has to be invoked. You can use this attribute in two different ways. The first way is to use '*' with the value of this attribute. Using '*' with the property attribute matches all the properties of a bean with the request parameter names. However, if you want to match any specific property of bean, you need to specify the value of the property attribute with the property name rather than using '*'. Consider the following syntax to use this attribute with '*' and with a specific property name:

```
// Syntax to use property attribute with '*'
<jsp:setProperty name=" name of reference variable " property="*" />
// Syntax to use property attribute with specific value
```

```
<jsp:setProperty name="name of reference variable" property="property name" />
```

Note that when you use '*' with the property attribute, the value and param attributes are not applicable with this tag. If you specify a specific property name for this attribute, the value and param attributes are applicable.

The values of the request parameters sent from the client to the server are always of type String. These values must be converted into the bean property types when stored in bean properties. If a property has a PropertyEditor class, as indicated in the JavaBeans specification, the setAsText(String) method is used to convert the value of the request parameter from String to a bean compatible data type. If the setAsText(String) method fails to convert the type, it throws an IllegalArgumentException.

The value Attribute

The value attribute takes the value that has to be set to the specified bean property. This attribute accepts the value as the String type or as an expression that is evaluated at run time. If the value is not of the String type, the value is converted to a Bean compatible data type.

NOTE

This attribute should not be used if property="", because it refers to all properties and we cannot set all properties using one value.*

The syntax to use the <jsp:setProperty> tag with the value attribute is as follows:

```
<jsp:setProperty name=" name of reference variable " property="property name" value= "
   Property value as String "/>
```

or

```
<jsp:setProperty name=" name of reference variable " property="property name" value= " <%=
   Property Value as expression %> "/>
```

The param Attribute

When setting bean properties from request parameters, it is not necessary that the bean has the same property names as the request parameters. The param attribute is used to specify the name of the request parameter whose value we want to assign to a bean property. If the param value is not specified, it is assumed that the request parameter and the bean property have the same names.

NOTE

This attribute must also not to be used when property="", because it refers to all properties and we cannot set requested parameters for all properties using one value.*

The syntax to use the param property is as follows:

```
<jsp:setProperty name="reference variable name" property="property name" param="request
   parameter name"/>
```

Let us see some examples to use the syntax given with attributes of the <jsp:setProperty> action tag, as given here:

- To set all properties in the JavaBean while setting bean properties from the request object:

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

- To set a specific property in the JavaBean while setting bean properties from the request object:

  ```
  <jsp:setProperty name="abcbean" property="uname" param="uname"/>
  ```

- When setting a bean property to a value, you should specify the value attribute as either a String or an expression that is evaluated at runtime:

  ```
  <jsp:setProperty name="abcbean" property="uname" value="<%=uname%>"/>
  ```

You learned about the <jsp:setProperty> action tag in this section. Let's now learn about the <jsp:getProperty> action tag next.

Describing the getProperty Tag

The <jsp:getProperty> action tag gets the value of a property in a bean by using the bean's getter methods and writes the value to the current JspWriter. The syntax to use the <jsp:getProperty> action tag is as follows:

```
<jsp:getProperty attributes/>
```

The <jsp:getProperty> action tag takes two attributes, name and property, which are described next.

The name Attribute

The name attribute takes the reference variable name on which you want to invoke the getter method. If a bean is instantiated by using the jsp:useBean action tag, the value assigned to the name attribute should match with the id attribute value of the jsp:useBean action tag. In this case, the <jsp:useBean> action tag must appear before the <jsp:setProperty> tag in the JSP page.

The property Attribute

The property attribute of the <jsp:getProperty> tag gets the value of a bean property and invokes the getter method of the bean property. This attribute takes the name of the property as an argument. For example, if the getUname method needs to be called, the property attribute takes uname, as shown in the following snippet:

```
<jsp:getProperty name="mybean" property="uname"/>
```

In the above code snippet, the <jsp:getProperty> tag gets the value of the uname property, by using the mybean instance, and includes this value into the output.

NOTE

<jsp:getProperty> is not designed to access Enterprise JavaBeans and indexed properties.

Plugin

The jsp:plugin action tag provides easy support for including a Java applet or bean in the client Web browser, using a built-in or downloaded Java plug-in. In the request handling stage of the JSP lifecycle, the jsp:plugin action tag is replaced by either the <object> or the <embed> tag, depending on the browser version. In general, the attributes of the jsp:plugin action tag perform the following operations:

- Specify whether the component added in the <object> tag is a bean or an applet
- Locate the code that needs to be run
- Position the object in the browser window
- Specify a URL from which to download the plug-in software
- Pass parameter names and values to the object

The following syntax shows the use of the JSP plugin action tag in a JSP page:

```
<jsp:plugin attributes>
<!-optionally one jsp:params and or one jsp:fallback tag can be used -->
</jsp:plugin>
```

The plugin tag takes some predefined attributes, which are:

- **type**—Specifies the type of object that needs to be presented to the browser. The object can be an applet or a JavaBean.
- **code**—Takes the qualified class name of the object that has to be presented.
- **codebase**—Takes the base URL where the specified class can be located. This attribute is optional.
- **name**—Specifies the name of the instance of the bean or applet, which helps the applets or beans called by the same JSP page to communicate with each other.
- **archive**—Specifies a comma-separated list of pathnames, which locate archive files that are preloaded with a class loader located in the directory named 'codebase'. The archive files are loaded securely, often over a network, and typically improve the applet's performance. This attribute is similar to the archive attribute of the HTML applet tag.
- **width**—Specifies the initial width, in pixels, of the image the applet or bean displays.
- **height**—Specifies the initial height, in pixels, of the image the applet or bean displays.
- **align**—Specifies the position of the applet. The values that the align attribute can take are bottom, top, middle, left, and right. The default value is bottom.

- **hspace**—Specifies the amount of space, in pixels, to the left and right of the applet or bean displayed by the browser. The value must be a nonzero number.
- **vspace**—Specifies the amount of space, in pixels, to the bottom and top of the applet or bean. The value must be a nonzero number.
- **jreversion**—Specifies the version of the Java Runtime Environment (JRE) that the corresponding applet or bean requires. The default value is 1.2.
- **nspluginurl**—Specifies the URL where the client can download the JRE plug-in for Netscape Navigator. The value of this attribute is the full URL specifying a protocol name, port number (this is optional), and domain name.
- **iepluginurl**—Specifies the URL where the client can download the JRE plug-in for Internet Explorer. The value of this attribute is the full URL specifying a protocol name, port number (optional), and domain name.

Params

The params action tag sends the parameters that we want to pass to an applet or bean. The following code snippet shows the syntax of using the params action tag in a JSP page:

```
<jsp:params>
<!- one or more jsp:param tags-->
</jsp:params>
```

To specify more than one parameter value, we can use more than one jsp:param action tag within a jsp:params tag.

Fallback

The fallback action tag allows us to specify a text message to be displayed if the required plug-in cannot run. This action tag must be used as a child tag with the jsp:plugin action tag. If the plug-in runs but the applet or bean does not, the plug-in usually displays a popup window explaining the error to the user. The following syntax shows the use of the fallback action tag in a JSP page:

```
<jsp:fallback>
Text message that has to be displayed if the plugin cannot be started
</jsp:fallback>
```

Attribute

The jsp:attribute action tag can be used to specify the value of a standard or custom action attribute. For example, we can use the jsp:attribute tag to set the attributes of the jsp:setProperty attribute, as shown in the following code snippet:

```
<jsp:setProperty name="mybean">
  <jsp:attribute name="property">uname</jsp:attribute>
  <jsp:attribute name="value">
        <jsp:expression>uname</jsp:expression>
  </jsp:attribute>
</jsp:setProperty>
```

The jsp:attribute tag accepts the name and trim attributes; where *name* specifies the attribute name that we want to set, as shown in the preceding sample code; and *trim* takes true or false, indicating whether or not the whitespace appearing at the beginning and at the end of jsp:attribute tag should be discarded. By default, the leading and trailing whitespaces are discarded; consequently, the default value of the trim attribute is true.

NOTE

The JSP container performs this operation at transaction time and not at request handling phase. For example, if we use an expression tag in the body part of this action tag, and the expression tag produces a value with leading and trialing whitespaces, then these whitespaces are not discarded by the JSP container. In the preceding code snippet, if the uname variable contains leading and trailing whitespaces then container does not eliminate the whitespaces.

If the body of the jsp:attribute tag is empty, you can specify its value by "".

Body

The <jsp:body> action tag can be used to specify the content (or the body) of a standard or custom action tag. Generally, the body of a standard or custom action invocation is implicitly defined as the body of that tag. However, if one or more <jsp:attribute> tags appear in the body of the tag, the body of a standard or custom action can be defined explicitly by using the <jsp:body> tag. The jsp:body tag is required if the standard action or custom tag has multiple jsp:attribute tags.

You can use the jsp:body tag to specify the content for the body of JSP actions tags, except for some action tags such as jsp:body, jsp:attribute, jsp:scriptlet, jsp:expression, and jsp:declaration.

If one or more jsp:attribute tags appear in the body of a tag invocation but no or empty jsp:body tag appears, it is equivalent of the tag having an empty body. The following code snippet shows the usage of the <jsp:body> tag:

```
<jsp:useBean id="mybean">
  <jsp:attribute name="class" trim="true">
      com.kogent.jspex.MyBean
  </jsp:attribute>
  <jsp:attribute name="scope">session</jsp:attribute>
  <jsp:body>
      <jsp:setProperty name="mybean" property="*"/>
  </jsp:body>
</jsp:useBean>
```

Element

The jsp:element action tag is used to dynamically define the value of the tag of an XML element. This action tag allows you to create an XML tag with a given name. The content of the jsp:element tag is a template for the attributes and child nodes of the XML tag you want to create. This action tag can be used in JSP pages and tag files. The following code snippet shows the syntax of the element tag:

```
<jsp:element name="name">
  jsp:attribute*
  jsp:body?
</jsp:element>
```

The *name* attribute of the element tag specifies the name of the XML element it has to create. You can also give an expression as the name of the XML element. The jsp:element tag can be an empty tag or can contain one jsp:body or multiple jsp:attribute attributes; with or without the jsp:body tag. The code snippet below shows how to use the name attribute with the element action tag:

```
<jsp:element name="mytag"/>
```

The <jsp:element> shown in the above code snippet creates an empty tag with the name, *mytag*.

The following code snippet shows another example of using the <jsp:element> tag with the <jsp:attribute> tag:

```
<jsp:element name="mytag">
  <jsp:attribute name="myatt">Myval</jsp:attribute>
</jsp:element>
```

The <jsp:element> tag shown in the above code snippet, creates an empty element, named *mytag*, with an attribute *myatt="Myval"*.

The following code snippet shows the use of <jsp:tag> to create an tag with attribute and body content:

```
<jsp:tag name="mytag">
  <jsp:attribute name="myatt">Myval</jsp:attribute>
  <jsp:body>Hello</jsp:body>
</jsp:tag>
```

The preceding code snippet creates the tag named *mytag* with an attribute *myatt="Myval"* and body text content as *Hello*.

Text

A jsp:text tag is used to enclose the template data in an XML tag. The text tag can be used in a JSP page or Tag file. The content of a jsp:text tag is passed to the out implicit object. A jsp:text tag has no attributes and can appear at any place where template data appears. The syntax to use the Text tag is:

```
<jsp:text> template data </jsp:text>
```

Note that the template data can also contain expressions.

In this section, you learned about the new JSP standard action tags introduced in JSP 2.0. Let's now learn how to declare a bean in JSP.

Using a Bean in a JSP

Before declaring a Bean in a JSP, you must create a Bean. Let's look at an example to create a Bean. In this example, we create a Bean called RegForm, which is used to set and get the attributes (such as user name and email address) of a user. Let's create the useBeanEx application to demonstrate how to use a JavaBean in a JSP page. You can find the code files of useBeanEx application on the CD in the code\Java EE\Chapter 12\ useBeanEx folder.

Follow these broad-level steps to declare a bean in a JSP:

1. Create a Bean
2. Declare the Bean in a JSP by using the <jsp:useBean> tag
3. Access the Bean properties
4. Generate Dynamic Content
5. Deploy and run the application

Creating a Bean

We start by creating a Bean called RegForm. Listing 12.25 shows how to create the RegForm Bean (you can find the RegForm.java file in the code\Java EE\Chapter 12\ useBeanEx\src\com\kogent folder on the CD):

Listing 12.25: The RegForm.java File

```
package com.kogent.jspex;
public class RegForm implements java.io.Serializable {
    private String uname, pass, repass, email, fn, ln, address;
    public void setUserName(String s){uname=s;}
    public void setPassword(String s){pass=s;}
    public void setRePassword(String s){repass=s;}
    public void setEmail(String s){email=s;}
    public void setFirstName(String s){fn=s;}
    public void setLastName(String s){ln=s;}
    public void setAddress(String s){address=s;}
    public String getUserName(){return uname;}
    public String getPassword(){return pass;}
    public String getRePassword(){return repass;}
    public String getEmail(){return email;}
    public String getFirstName(){return fn;}
    public String getLastName(){return ln;}
    public String getAddress(){return address;}
}//class
```

Declaring the Bean in a JSP

After creating the Bean, you can declare it in a JSP page by using the <jsp:useBean> tag. To declare the Bean in a JSP, you have to create a JSP page. In Listing 12.26, we have created a JSP page with the name RegProcess, in which we declare the bean created in Listing 12.25. Listing 12.26 shows the code for the RegProcess.jsp file (you can find the RegProcess.jsp file in the code\Java EE\Chapter 12\useBeanEx folder on the CD):

Listing 12.26: The RegProcess.jsp File

```
<%@page errorPage="Registration.html"%>
<html>
<body>
<jsp:useBean id="regform" class="com.kogent.jspex.RegForm" scope="session"/>
<jsp:setProperty name="regform" property="*"/>
<form action="RegProcessFinal.jsp"><pre> <b>
First Name  : <input type="text" name="first_name"/>
Last Name   : <input type="text" name="last_name"/>
Address     : <input type="text" name="address"/>
          <input type="submit" value="Register"/>
</b></pre></form>
</body>
</html>
```

The <jsp:useBean> action tag creates an instance of a bean class according to the attributes specified in the JSP page. The attributes of the <jsp:useBean> tag used here are:

- `id`—Specifies the name of the bean that you want to declare
- `class`—Specifies name of the Bean class, which helps the JSP container in searching the Bean class and creating its instance.
- `scope`—Specifies the scope of the bean

All these attributes help JSP container to create an instance of the Bean class. The <jsp:useBean> action tag has some more attributes along with the attributes used in. You can also specify these attributes to make the Bean more specific.

Accessing the Bean Properties

You can verify the bean properties by using the <jsp:getProperty> action tag. The <jsp:getProperty> action tag uses the name and property tags to read the Bean properties. We have created the ViewRegistrationDetails.jsp page in this application to demonstrate how the <jsp:getProperty> action tag is used to read the Bean property, as shown in Listing 12.27. The code for the ViewRegistrationDetails page is provided in Listing 12.27 (you can find the ViewRegistrationDetails.jsp file in the code\Java EE\Chapter 12\useBeanEx folder on the CD):

Listing 12.27: The ViewRegistrationDetails.jsp File

```
<jsp:useBean id="regform" type="com.kogent.jspex.RegForm" scope="session"/>
<%@page errorPage="Registration.html"%>
<html>
<body>
<pre>
<b>User Name   :</b> <jsp:getProperty name="regform" property="userName"/>
<b>Password    :</b> <jsp:getProperty name="regform" property="password"/>
<b>Email ID    :</b> <jsp:getProperty name="regform" property="email"/>
<b>First Name :</b> <jsp:getProperty name="regform" property="firstName"/>
<b>last Name   :</b> <jsp:getProperty name="regform" property="lastName"/>
<b>Address     :</b> <jsp:getProperty name="regform" property="address"/>
</pre>
<form method=post action="javascript:alert('The remaining process is under
  development');">
<input type="submit" value="Register"/>
</form>
</body>
</html>
```

Before reading the Bean properties, the JSP container needs to locate the instance of the Bean class created by you. After the JSP container locates the instance of the Bean class, it starts reading the Bean properties by using the <jsp:getProperty> tag. In Listing 12.27, you can see the attributes used with the <jsp:getProperty> tag. The name attribute specifies the name of the Bean class to the JSP container and the property attribute specifies the Bean properties that have to be read.

Generating Dynamic Content within a JSP

The next step is to create an HTML page, named Registration.html, to retrieve the required data from users. The Registration.html page contains a button called Register. When a user clicks the Register button after providing all user details, he/she is directed to the RegProcess.jsp page to set the properties of 'RegForm' JavaBean, based on the details entered by the user. In other words, we use the Registration.html page to generate dynamic content in the RegProcess.jsp page. Listing 12.28 shows how to create the Registration.html page (you can find the Registration.html file in the code\Java EE\Chapter 12\useBeanEx folder on the CD):

Listing 12.28: The Registration.html File

```
<html>
<body>
<pre>
<form action="RegProcess.jsp"><pre><b>
UserName   : <input type="text" name="userName"/>
Password   : <input type="password" name="password"/>
RePassword : <input type="password" name="rePassword"/>
EMail ID   : <input type="text" name="email"/>

          <input type="submit" value="Register"/>
</b></pre></form>
</pre>
</body>
</html>
```

In the Registration.html page, the param names such as userName, password, and rePassword are the same as the property names in the RegForm bean. However, the other param names, such as first_name and last_name, are different from the property names such as firstName and lastName. Therefore, the RegProcess JSP page is created to set these properties in the RegForm bean.

Using this page, you can declare a bean in JSP. The RegProcessFinal JSP page also collects some more information about the user in the RegForm JavaBean in addition to what we have collected by using the Registration.html page. Here also, a **Register** button is provided, when you click on that button after entering all user information, you are directed to the RegProcessFinal. JSP page, as shown in Listing 12.29 (you can find the RegProcessFinal.jsp file in the code\Java EE\Chapter 12\useBeanEx folder on the CD):

Listing 12.29: The RegProcessFinal.jsp File

```
<%@page errorPage="Registration.html"%>
<jsp:useBean     id="regform"
type="com.kogent.jspex.RegForm" scope="session"/>
<jsp:setProperty name="regform" property="firstName" param="first_name"/>
<jsp:setProperty name="regform" property="lastName" param="last_name"/>
<jsp:setProperty name="regform" property="address"/>
<html>
<body>
<pre>
Your registration details are valid,
<a href="ViewRegistrationDetails.jsp">Click</a> to view Registration Details and confirm.
</pre>
</body>
</html>
```

NOTE

Before using the <jsp:setProperty> or <jsp:getProperty> tag in a JSP page, you must declare a bean in that JSP page by using the <jsp:useBean> tag. Before deploying and running the application, ensure that you have compiled all the Java source files.

Deploying and Running Application

You must place the pages and other resources as per the correct directory structure of this application. Figure 12.20 shows the directory structure of this application:

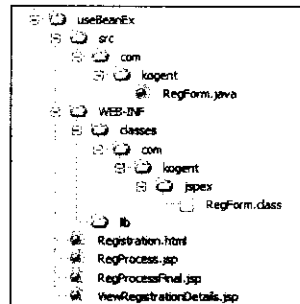

Figure 12.20: Displaying the Directory Structure of the useBeanEx Application

Copy the userBeanEx folder into the <tomcat home>/webapps folder and start the Tomcat server.

The Deployment Descriptor web.xml shown in the directory structure contains an empty </web-app> tag.

Now, browse the application by using the URL http://localhost:9090/userBeanEx/Registration.html and enter the details as shown in Figure 12.21:

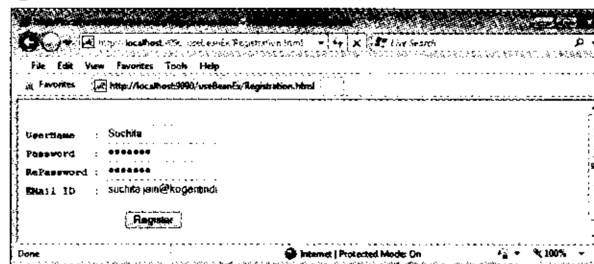

Figure 12.21: Displaying the Registration window.

Click the Register button to continue with the registration process. In this example, we store the username, password, and email id into the RegBean by using the useBean and setProperty tags. We also set the RegBean instance in the session scope. After you enter the details and click the Register button in the first screen, as shown in Figure 12.21, the next screen displays the information of the user, as shown in Figure 12.22:

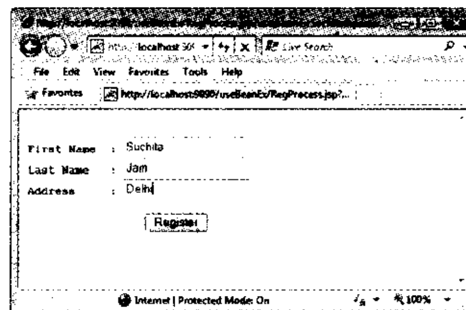

Figure 12.22: Displaying the Registration Process window.

When you click the Register button of the second screen, the first name, last name, and address values are also stored into the RegBean. The next screen confirms the validity of the registration details, as shown in Figure 12.23:

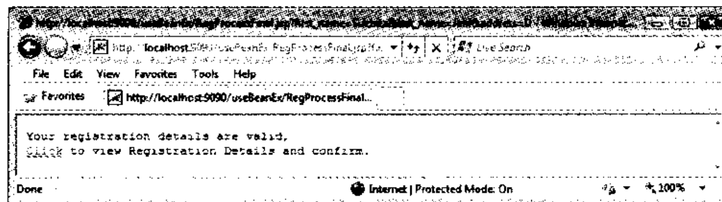

Figure 12.23: Registration final process window.

Clicking the hyperlink shown in Figure 12.23 displays all the details entered in the first and second screens, as shown in Figure 12.24:

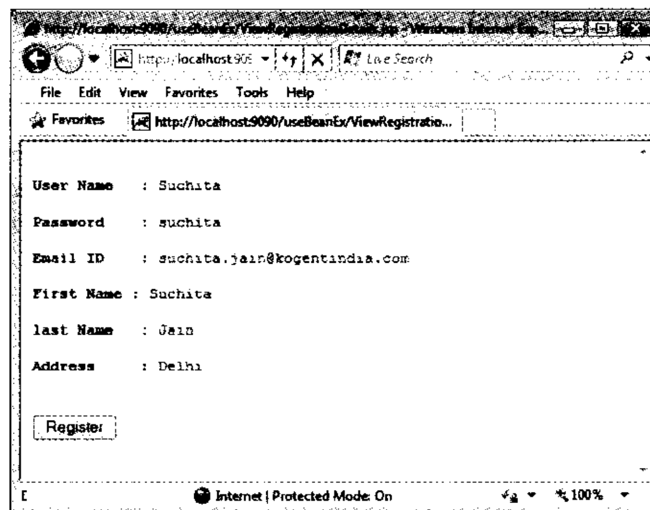

Figure 12.24: Registration Detail window.

This screen confirms that the RegBean has been successfully created and set.

After understanding how to use the JavaBeans in a JSP page, let's now discuss the implementation of JSP standard tag library.

Using the JSP Standard Tag Library (JSTL)

The JSP Standard Tag Library (JSTL) is a collection of custom tag libraries, which provide core functionality used for JSP documents. JSTL reduces the use of scriptlets in a JSP page. The use of JSTL tags allows developers to use predefined tags instead of writing the Java code. JSTL provides four types of tag libraries that can be used with JSP pages:

- **The Core JSTL**—Used to process a JSP page in an application.
- **The XML Tag library**—Used for parsing, selecting, and transforming XML data in a JSP page.
- **The Format Tag library**—Used for formatting the data used in a JSP page.
- **The SQL Tag library**—The SQL tag library is used to access the relational database used in a JSP page.

These tag libraries are described in detail next.

Describing JSTL Core Tags

The JSTL core tag library provides the JSTL core tags used in the JSP pages. The core tags are used to perform iteration, conditional processing, and also provide support of expression language for the tags in JSP pages. The core tag library can be used in a JSP page by accessing the following tag library:

```
< %@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

The core tags in a JSP page can be accessed by using the prefix 'c', the preferred prefix for core tag libraries.

The JSTL core tags are divided into three categories as listed here:

- General-purpose Tags
- Conditional and Looping Tags
- Networking Tags

The following section explains all these tags categories in detail. First, we start with general-purpose tags.

General-Purpose Tags

General-purpose tags include tags for writing the values to the output stream, as well as retrieving, setting, and removing attributes and also include tags for catching exceptions. The general-purpose tags contain four tags:

- <c:out>
- <c:set>
- <c:remove>
- <c:catch>

Let's now discuss these tags.

The <c:out> tag

The <c: out> tag evaluates an expression that may be given in its `value` attribute or in its tag body and the resulting value is written in the output. The syntax of the <c: out> tag is as follows:

```
<c:out attributes> [body content] </c:out>
```

The attributes of the <c: out> tag are:

- **value**—Specifies the output data to be displayed.
- **escapeXml**—Specifies whether or not the tag should ignore the special XML characters. If boolean value of escapeXML is *true*, then the tag converts the XML special characters, such as <,>, & to their character entity equivalents, such as < > and &. If the value of escapeXml is false then it does not convert the XML special characters. The default value is true.
- **default**—Takes the value to be written to output if the given value resolves to null. If the value does not resolve to null, the value specified in the value attribute is displayed. The default attribute can be even specified in the body of the tag.

Using the <c:out> Tag

Let's create an application, jstlex1, which shows the use of the <c:out> tag. The application prints some request related data by using the <c:out> tag. Listing 12.30 shows the use the <c:out> tag (you can find the GetRequestData.jsp file in the code\Java EE\Chapter 12\ jstlex1 folder on the CD):

Listing 12.30: The GetRequestData.jsp File

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>

<html>

<body>

  <b>Requested URL:  </b><br/>
  <c:out value="${pageContext.request.requestURL}"/><br/>
  <br/><h4>Request Path information:</h4>
  <table border='1'>

        <tr><th>Name</th><th>Value</th></tr>
        <tr><td>HTTP Request method</td>
        <td><c:out value="${pageContext.request.method}"/></td></tr>
        <tr><td>Request URI</td>
```

```
            <td><c:out value="${pageContext.request.requestURI}"/></td></tr>
            <tr><td>Context Path</td>
            <td><c:out value="${pageContext.request.contextPath}"/></td></tr>
            <tr><td>Servlet path</td>
            <td><c:out value="${pageContext.request.servletPath}"/></td></tr>
            <tr><td>Path info</td>
            <td><c:out value="${pageContext.request.pathInfo}"/></td></tr>
            <tr><td>Path translated</td>
            <td><c:out value="${pageContext.request.pathTranslated}"/></td></tr>
    </table><br/>

    <h4>Accessing Request Parameter value:</h4>
    Value of parameter <b>uname</b> is:
    <b><c:out value="${param.uname}"/></b>

  </body>

  </html>
```

Listing 12.31 shows the web.xml file used for jstlex1example (you can also find the web.xml file on the CD in the code\Java EE\Chapter 12\jstlex1\WEB-INF folder on the CD):

Listing 12.31: The web.xml File

```
  <web-app>

    <welcome-file-list>
    <welcome-file>GetRequestData.jsp</welcome-file>
    </welcome-file-list>

  </web-app>
```

Now to run jstlex1 example, we need two jar files, the first one is the `jstl.jar`, which is used to support the JSTL API classes. The second is the `standard.jar`, which is used to implement classes of libraries. The jstl.jar and standard.jar files enable us to access the JSTL tag library. Both jstl.jar and standard.jar files are present in the Tomcat specification. To provide the support of JSTL in our Web application, copy the jstl.jar and standard.jar files into the lib directory of your application. Since the libraries are present in the lib directory of our Web application, so we do not have to update the Deployment Descriptor. The JSP container automatically detects and executes these JAR files.

After creating all the preceding files, arrange them, along with the jstl.jar and standard.jar files, in the directory structure shown in Figure 12.25:

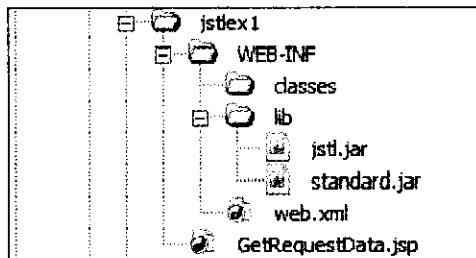

Figure 12.25: Showing directory structure of jstlex1 application

Copy the jstlex1 folder into the webapps folder of the Tomcat installation. Start the server and launch the application by using the http://localhost:9090/jstlex1/GetRequestData.jsp?uname=Kogent URL. The Requested URL window opens, as shown in Figure 12.26:

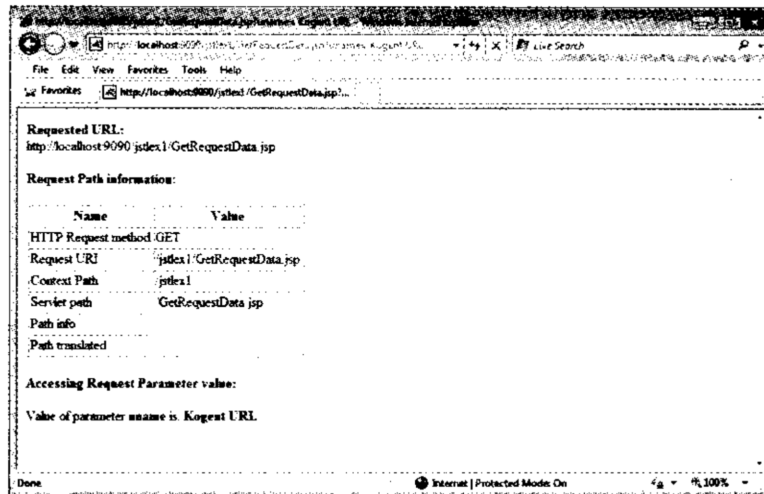

Figure 12.26: Showing the Output of GetRequestData.jsp.

Figure 12.26 shows the output of the GetRequestData.jsp file in which we have used the pageContext implicit object of Expression Language and retrieved the request data. We have also used the param object to get the parameter value.

The <c:set> Tag

The <c: set> tag is another general purpose tag, which allows us to set the value of a variable or property into the given scope. The following code snippet shows the syntax of the <c: set> tag:

```
<c:set attribute> [body content] </c:set>
```

The attributes of the <c: set> tag are listed here:

- **value** – Specifies the value being set. The value can also be an expression.
- **var** – Specifies the name of the variable to store.
- **scope** – Specifies the scope of the variable defined in the var attribute. Scope can be page, request, session, or application.
- **target** – Specifies the name of the variable whose property is to be set.
- **property** – Specifies the name of the property to be set

Note that if target attribute is specified, the property attribute must also be specified. In addition, you can use either the var or target attribute for specifying the name of the variable.

The <c:remove> Tag

The <c: remove> general-purpose core JSTL tag allows us to remove a variable from the specified scope. The syntax of the <c: remove> tag is:

```
<c:remove attributes/>
```

The <c: remove> tag takes two attributes, which are:

- **var** – Name of the variable to be removed.
- **scope** – Scope of the variable, scope can be page, request, session, or application.

The <c:catch> Tag

The <c:catch> tag is used to allow JSP pages to handle exceptions that might be raised by any of the tag inside the body of the <c:catch> tag. The syntax of the <c: catch> tag is:

```
<c:catch attributes> body content </c:catch>
```

The <c: catch> tag accepts the var attribute, which takes the name of the variable that stores the exception thrown.

Let's now describe the conditional and looping tags of JSTL.

Conditional and Looping Tags

The conditional and looping tags include tags for writing if conditions, switch cases, and loops.

Types of Conditional and Looping Tags

Conditional and looping tags are of six types:

- <c: if>
- <c: choose>
- <c: when>
- <c: otherwise>
- <c: forEach>
- <c: forToken>

The following sections explain all these tags in detail.

The <c:if> Tag

The <c:if> tag evaluates an expression that is given in its 'test' attribute. If the resulting value is *true*, the body of the <c:if> tag is evaluated; otherwise not. The following snippet shows the syntax of the <c:if> tag of JSTL:

```
<c:if attributes> body content </c:if>
```

The attributes of <c: if> tag are listed here:

- **test** – The boolean variable or expression resolving to boolean value
- **var** – Name of the variable to store the conditions result
- **scope** – Scope of the variable to store the conditions result

The following code snippet shows the example for <c: if> tag:

```
<c:if test="${param.uname==''}">
  Username cannot be empty
</c:if>
```

In the preceding shown snippet, the <c:if> tag checks whether or not the value of request parameter 'uname' is an empty string. If it is an empty string, the body part of the <c:if> tag is evaluated; if not, the body part is skipped. Note that we do not have a tag that allows us to specify else part for the <c:if> tag. If we want to implement the else part of the <c:if> tag, then use <c:if> once again. The following section explains the <c: choose> tag.

The <c:choose> Tag

The <c:choose> tag acts like a Java switch statement. The <c: choose> tag encloses one or more <c: when> tags and a <c:otherwise> tag. The syntax of the <c:choose> tag is:

```
<c:choose> body content </c:choose>
```

The following section explains about the <c:when> tag.

The <c:when> Tag

The <c:when> tag encloses a single case within the <c:choose> tag. The <c:when> tag must appear inside the <c:choose> element. More than one <c:when> tag can be defined in a single <c:choose> element.

The syntax of the <c:when> tag is:

```
<c:when attribute> body content </c:when>
```

The <c:when> tag accepts only one attribute, named test. The test attribute takes the boolean variable or an expression resolving to a boolean value. If this boolean value is true, the body content of the <c:when> tags is evaluated. If this value is false, then the body content of the <c:when > tag is skipped.

The <c:otherwise> Tag

The <c:otherwise> is same as the <c: when> tag, but is unconditional. That is, the <c: otherwise> tag is equivalent to the default case in a Java switch statement. The body content of the <c: otherwise> tag is evaluated if none of the <c: when> conditions in the <c: choose> tag are resolved to true.

The syntax of the <c: otherwise> tag is:

```
<c:otherwise> body content </c:otherwise>
```

Now, after learning about the <c:otherwise> tag, let's look at <c: forEach> tag.

The <c: forEach> Tag

The <c: forEach> tag is used to iterate over a collection of objects or as a fixed loop over a range of numbers. A common use of the <c:forEach> tag is to produce a HTML table containing data gathered from a SQL query or other data source.

The syntax of the <c: forEach> tag of JSTL is:

```
<c: forEach attributes> body content </c: forEach>
```

The attributes of the <c: forEach> are:

- **items**—Specifies the collection to iterate over. The collection can be ArrayList, Map, or List.**var**: Specifies the name of the variable into which the current iteration item has to be set
- **varStatus**—Specifies the name of the variable that defines the status of the variable given in the var attribute. The varStatus attribute is optional.
- **begin**—Takes the starting index for the loop; this attribute is optional
- **end**—Takes the ending index for the loop; this attribute is optional
- **step**—An optional increment for the loop; the default value is 1

The <c:forTokens> Tag

The <c:forTokens> tag is used for looping over tokenized elements of a string.

The syntax of the <c:forTokens> tag of JSTL is:

```
<c:forTokens attributes> body content </c:forTokens>
```

The attributes of the <c: forTokens> tag are:

- **items**—Specifies the String to be tokenized. This attribute allows an expression
- **var**—Specifies the name of the variable into which the current iteration item has to be set
- **delims**—Specifies the delimiter or delimiters, which are to be tokenized.
- **varStatus**—Defines the status of the variable given in var attribute, The varStatus attribute is optional
- **begin**—Takes the starting index for the loop; this attribute is optional
- **end**—Takes the ending index for the loop; this attribute is optional
- **step**—Provides an optional increment for the loop; the default value is 1

Using Conditional and Looping Tags

We demonstrate two examples to use conditional and looping tags. The first example uses the <c:if>, <c:when>, <c:choose>, and <c:otherwise> tags. The second example uses the <c:forEach> tag.

An Example of Using Conditional and Looping Tags

Let's create the jstlex2 application that demonstrates the use of conditional tags. In this application, we create two pages, index.html and ProcessRequest.jsp.

The index.html page contains two text boxes, to accept two operands from the users, and two buttons named Add and Subtract to demonstrate the add and subtract operations on the operands. Listing 12.32 shows the code for the index.html file (you can find index.html file in the code\Java EE\Chapter 12\jstlex2 folder on the CD):

Listing 12.32: The index.html File

```
<html>
  <body>
```

```
            <form action="ProcessRequest.jsp"><pre>
                   Operand 1 <input type="text" name="op1"/>
                   Operand 2 <input type="text" name="op2"/>
                   <input type="submit" name="submit" value="Add"/>
                   <input type="submit" name="submit" value="Subtract"/>
            </form></pre>
      </body>
   </html>
```

When the user clicks the Add or Subtract button on the index.html page, the control is transferred to the ProcessRequest.jsp page. This page takes the operand from the index.html page. The JSP page uses the <c: if> tag to check the condition. The condition is either add or subtract, and depending on the condition, the required operation is performed. The output is displayed by using <c: out>tag. Listing 12.33 provides the code for the ProcessRequest.jsp page (you can find the ProcessRequest.jsp file in the code\Java EE\Chapter 12\jstlex2 folder on the CD):

Listing 12.33: The ProcessRequest.jsp File

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<html>
<body>
   <c:if test="${param.submit eq 'Add'}">
   Sum of <c:out value="${param.op1}"/> and <c:out value="${param.op2}"/> is <c:out
   value="${param.op1 + param.op2}"/>
   </c:if>
   <c:if test="${param.submit eq 'Subtract'}">
   <c:out value="${param.op1}"/> - <c:out value="${param.op2}"/> = <c:out
   value="${param.op1 - param.op2}"/>
   </c:if>
</body>
</html>
```

After creating the two pages, we need to configure the application by using the web.xml file. Listing 12.34 contains the web.xml file to configure the jstlex2 application (you can find the web.xml file in the code\Java EE\Chapter 12\jstlex2\WEB-INF folder on the CD):

Listing 12.34: The web.xml File

```
<web-app>
   <welcome-file-list>
          <welcome-file>index.html</welcome-file>
   </welcome-file-list>
</web-app>
```

After creating all the preceding files, arrange them, along with the jstl.jar and standard.jar files, as shown in Figure 12.27:

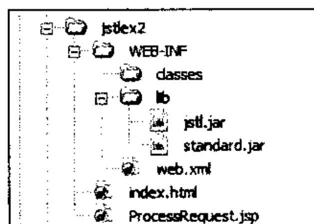

Figure 12.27: Showing Directory Structure of the jstlex2 Application.

Copy the jstlex2 folder into the webapps folder of the Tomcat installation and start the Tomcat server.

After starting the server, launch the application by using the following URL: `http://localhost:9090/jstlex2/index.html`. Figure 12.28 shows the output of the index.html page:

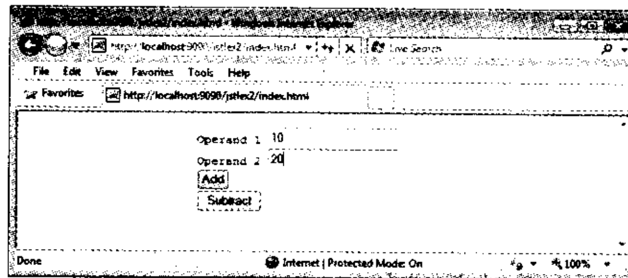

Figure 12.28: Showing index.html to Enter the Operands

Enter some numbers into the text fields— operand 1 & operand 2—as shown in Figure 12.28 and click the required arithmetic operation, Add or Subtract. After entering the operand, when a user clicks the Add or Subtract button, the control is transferred to the ProcessRequest.jsp page. In our case, we have clicked the Add button, and the ProcessRequest.jsp page demonstrates the output of the add operation, as shown in Figure 12.29:

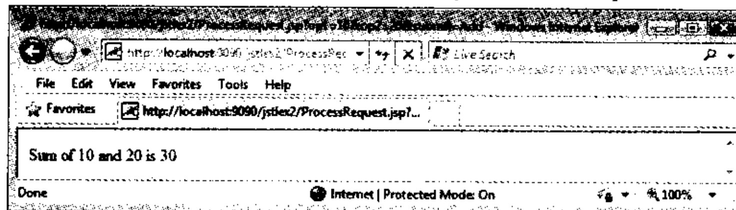

Figure 12.29: Showing Addition of Two Numbers

In the preceding example, we can use the <c:choose> tag instead of using two <c:if> tags, as shown in the following code snippet:

```
<c:choose>
  <c:when test="${param.submit eq 'Add'}">
  Sum of <c:out value="${param.op1}"/> and <c:out value="${param.op2}"/> is
  <c:out value="${param.op1 + param.op2}"/>
  </c:when>
  <c: otherwise>
       <c:out value="${param.op1}"/> - <c:out value="${param.op2}"/> =
       <c:out value="${param.op1 - param.op2}"/>
  </c:otherwise>
</c:choose>
```

Using the <c: forEach> Tag

The jstlex3 application demonstrates to get an employee information from the database and represents it to the user with a JSP page using the <c: forEach> tag. The <c:forEach> tag is used to display each employee details. In the jstlex3 application, GetEmpDetailsServlet.java is created to retrieve the data from the database using DriverConnection.java. The DriverConnection.java is used to create connection to MySQL data source. The EmpDetailsView.jsp displays the data retrieved by the servlet GetEmpDetailsServlet.java.

To get the data from the database, we use a servlet named GetEmpDetailsServlet.java, a Java bean named EmpDetails.java and a connection class named DriverConnection.java. To represent the Employee's details to the user, we use a JSP page (EmpDetailsView.jsp) in which we use <c: forEach> tag. Listing 12.35 shows the code for EmpDetails.java, where Java bean is used in our application to set and get data from the database. Listing 12.35 shows the code for the EmpDetails.java file (you can find the EmpDetails.java file in the code\Java EE\Chapter 12\jstlex3\src\com\kogent\jstl folder on the CD):

Listing 12.35: The EmpDetails.java File

```
package com.kogent.jstl;
import java.io.*;
public class EmpDetails implements Serializable {
```

```
    public EmpDetails (){}
    public EmpDetails (int i, String s, int j, int d){
        empno=i; name=s; deptno=j; sal=d;
    }
    public void setEmpNo(int i){empno=i;}
    public int getEmpNo(){return empno;}
    public void setName(String s){name=s;}
    public String getName(){return name;}
    public void setDeptNo(int i){deptno=i;}
    public int getDeptNo(){return deptno;}
    public void setSal(int d){sal=d;}
    public int getSal(){return sal;}
    String name;
    int empno,deptno, sal;
}//class
```

Listing 12.36 shows the code for DriverConnection.java, which is used to connect to the MySQL database (you can find the DriverConnection.java file in the code\Java EE\Chapter 12\jstlex3\src\com\kogent\jstl folder on the CD):

Listing 12.36: The DriverConnection.java File

```
package com.kogent.jstl;
import java.sql.*;
public class DriverConnection {
  public static Connection getConnection() throws Exception{
      Class.forName("com.mysql.jdbc.Driver");
      return DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/sample","root","root");
  }
}
```

Listing 12.36 shows the source code of the GetEmpDetailsServlet.java file. The GetEmpDetailsServlet.java file is used to retrieve the employee details from the database table (emp) into a result set, which is then stored in the array list. The retrieved details are then forwarded to the EmpDetailView.jsp (see Listing 12.38) through RequestDispatcher. Listing 12.37 shows the code for the GetEmpDetailsServlet class (you can find the GetEmpDetailsServlet.java file in the code\Java EE\Chapter 12\jstlex3\src\com\kogent\jstl folder on the CD):

Listing 12.37: The GetEmpDetailsServlet.java File

```
package com.kogent.jstl;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.sql.*;
public class GetEmpDetailsServlet extends HttpServlet {
  public void service(HttpServletRequest request,HttpServletResponse response)
  throws ServletException, IOException {
      Connection con=null;
      try {
            con=DriverConnection.getConnection();
            Statement st=con.createStatement();
            ResultSet rs=st.executeQuery("select * from emp");
            ArrayList al=new ArrayList();
            while (rs.next()){
                  al.add(new EmpDetails(
                  rs.getInt(1),
                  rs.getString("ename"),
                  rs.getInt("deptno"),
                  rs.getInt("sal")));
            }//while
```

```
                HttpSession session=request.getSession();
                session.setAttribute("emps",al);
                RequestDispatcher rd=
                request.getRequestDispatcher("EmpDetailsView.jsp");
                rd.forward(request, response);
                return;
            }//try
            catch(Exception e){e.printStackTrace();}
            finally{try{con.close();}
            catch(Exception e){e.printStackTrace();}}
    }
}
```

Listing 12.38 shows the source code of EmpDetailsView.jsp page, which is used to display the employee details forwarded through RequestDispatcher (see Listing 12.37). Listing 12.38 provides the code to display employee's details (you can find the EmpDetailsView.jsp file in the code\Java EE\Chapter 12\jstlex3 folder on the CD):

Listing 12.38: The EmpDetailsView.jsp File

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>

<html> <body>

  <table border="1">
        <tr><th>Empno</th>
        <th>Name</th><th>DeptNo</th>
        <th>Sal</th></tr>
        <c:forEach items="${sessionScope.emps}" var="emp">
              <tr> <td>
              <c:out value="${emp.empNo}"/>
              </td> <td>
              <c:out value="${emp.name}"/>
              </td> <td>
              <c:out value="${emp.deptNo}"/>
              </td> <td>
              <c:out value="${emp.sal}"/>
              </td> </tr>
        </c:forEach>
  </table>

</body>

</html>
```

Listing 12.39 describes the servlet mapping in web.xml file (you can find the web.xml file in the code\Java EE\Chapter 12\jstlex3\WEB-INF folder on the CD):

Listing: 12.39: The web.xml File

```
<web-app>
  <servlet>
        <servlet-name>ser1</servlet-name>
        <servlet-class>com.kogent.jstl.GetEmpDetailsServlet
        </servlet-class>
  </servlet>
  <servlet-mapping>
        <servlet-name>ser1</servlet-name>
        <url-pattern>/GetEmpDetails</url-pattern>
  </servlet-mapping>
</web-app>
```

After creating all the preceding files, arrange them along with jstl.jar, mysql-connector-java-5.0.4-bin.jar, and standard.jar as the directory structure shown in Figure 12.30:

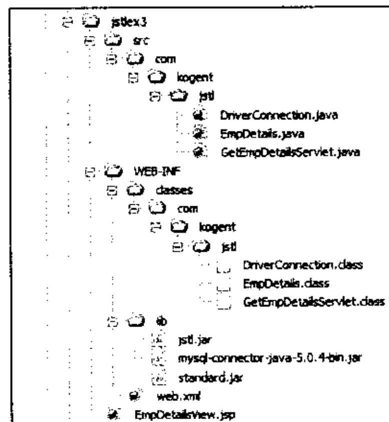

Figure 12.30: Showing Directory Structure of the jstlex3 Application

Now, copy the jstlex3 folder into webapps folder of the Tomcat installation and start the server.

NOTE

Before running the jstlex3 application, ensure that created the emp table in the MySQL database.

Browse the example by using the URL http://localhost:9090/jstlex3/GetEmpDetails. Figure 12.31 shows the output of the jstlex3 application:

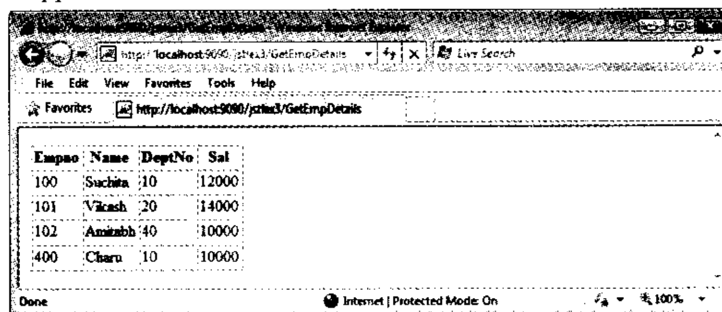

Figure 12.31: Showing Employee's detail

The application displays all employee's details as shown in Figure 12.31.

Networking Tags

Networking tags contain tags that perform certain operations on URLs, such as including some other Web pages, encoding the URL, and redirecting the client request. The networking tags of JSTL tags contains four tags:

- <c: import>
- <c: url>
- <c: redirect>
- <c:param>

The following section explains all these tags in detail.

The <c:import> Tag

The <c: import> tag is used to include another resource, such as another JSP or HTML page, in a JSP page. Moreover, the resource can be either static or dynamic. The <c: import> tag is similar to the <jsp: include> tag. However, the <c: import> tag allows you to include the pages in another Web application.

The following is the syntax of the <c:import> tag of JSTL:

```
<c: import attributes> </c: import>
```

The attributes of the <c: import> tag are:

- **url**—Specifies the URL of the resource to include
- **context**—Specifies the context name in which the page has to be located, this is an optional attribute
- **var**—Specifies the name of the variable into which the result has to be stored, if specified
- **scope**—Specifies the scope into which the variable has to be stored

The <c:url> Tag

The <c:url> tag creates a URL and is added with a session ID if the user session needs to be maintained. This tag functions similar to the encodeURL () method of the HttpServletRequest interface.

The following is the syntax of the <c:url> tag of JSTL:

```
<c:url attributes> [zero or more <c:param> tags] </c:url>
```

The attributes of the <c:url> tag are:

- **value**—Specifies the URL to be rewritten if required added with session ID
- **context**—Specifies the context name of another Web application, required if the URL refers to the resource in another servlet context
- **var**—Specifies the name of the variable into which the new rewritten URL has to be stored
- **scope**—Specifies the scope of the variable defined in var attribute.

The <c:redirect> Tag

The <c:redirect> tag is used to redirect a client request. This tag acts similar to the sendRedirect() method of the HttpServletResponse interface.

The following is the syntax of the <c:redirect> tag of JSTL:

```
<c:redirect attributes> [zero or more <c:param> tags] </c: redirect>
```

The attributes of the <c: redirect> tag are:

- **value**—Specifies the URL of the resource to which the request has to be redirected
- **context**—Specifies the context name of another Web application, required if the URL refers to the resource in another servlet context

The <c:param> Tag

The <c:param> tag is used to add a request parameter to the URL. This tag can be used in <c:url> and <c:redirect> tags.

The following is the syntax of the <c:param> tag of JSTL:

```
<c:param attributes/>
```

The attributes of the <c:param> tag are:

- **name**—Specifies the name of the parameter
- **value**—Specifies the parameter value

Describing the JSTL SQL Tags

The SQL tag library is used to access the relational database used in the JSP pages. The SQL tags are used for rapid prototyping and developing Web applications. The SQL tag libraries can be accessed in a JSP page by importing the following tag library in the JSP page:

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
```

Types of JSTL SQL Tags

The SQL tag library provides certain tags that are used in a JSP page to access a database. Table 12.2 shows the different SQL tags available in the JSTL SQL Tag libraries:

Table 12.2: The SQL Tag Library

Tag Name	Description
query	Executes a query specified in a JSP page
update	Updates an SQL statement used in a JSP page
param	Specifies a parameter in the SQL statement
dateParam	Sets a parameter in the SQL statement into a java.util.Date value
setDataSource	Specifies a data source that is to be accessed by executing a query
transaction	Provides a connection to all the database operations that are to be executed in a JSP page

Now, we discuss these SQL tags in detail in the next sections.

The <sql:query> Tag

The <sql:query> tag executes the query specified in the sql attribute or in the tag body. Then, the result of the query is set to the variable specified in the var attribute.

The following is the syntax of the <sql:query> tag of JSTL:

```
<sql:query attributes> [body content] </sql:query>
```

The attributes of the <sql:query> tag are:

- **sql**—Specifies the SQL query that has to be executed. This attribute is optional and if not given, the query needs to be specified in the tag body. The SQL query can be parameterized.
- **var**—Specifies the variable name to which the result of the query has to be set
- **scope**—Specifies the scope of the variable
- **dataSource**—Specifies the datasource JNDI name or java.sql.DataSource object
- **maxRows**—Specifies the maximum number of rows that has to be included into the result, this value needs to be greater than or equal to -1
- **startRow**—Specifies the starting row number

The body content for the <sql:query> tag accepts an SQL query and <sql:param> tags.

The <sql:update> Tag

The <sql:update> tag executes a SQL statement specified in the sql attribute or in the tag body. Then, the result of the query is set to the variable specified in the var attribute.

The following is the syntax of the <sql:update> tag of JSTL:

```
<sql:update attributes> [body content] </sql:update>
```

The attributes of <sql:update> tag are listed here:

- **sql**—Specifies the update SQL statement that has to be executed. This attribute is optional and if not given, the SQL statement needs to be specified in the tag body. The SQL query can be parameterized
- **var**—Specifies the variable name to which the result has to be set
- **scope**—Specifies the scope of the variable
- **dataSource**—Specifies the datasource JNDI name or java.sql.DataSource object

The <sql:param> Tag

The <sql:param> tag is used to set a parameter in the SQL statement.

The following is the syntax of the <sql:param> tag of JSTL:

```
<sql:param attributes>[parameter value]</sql:param>
```

The attribute of <sql:param> tag is **value**, which is used to specify the parameter value that has to be substituted in the SQL statement.

The <sql:dateParam> Tag

The <sql:dateParam> tag is used to set a Date parameter in the SQL statement.

The following is the syntax of the <sql:dateparam> tag of JSTL:

```
<sql:dateParam attributes/>
```

The attributes of the <sql:dateParam> tag are:

- **value**—Specifies the date parameter value that has to be substituted in the SQL statement
- **type**—Takes either 'date' or 'time' or 'timestamp'

The <sql:setDataSource> Tag

The <sql:setDataSource> tag binds a datasource to the specified variable.

The following is the syntax of the <sql:setDataSource> tag of JSTL:

```
<sql:setDataSource attributes/>
```

The attributes of the <sql:setDataSource> tag are:

- **dataSource**—Specifies the datasource JNDI name or java.sql.DataSource object
- **driver**—Specifies the JDBC driver class name
- **url**—Specifies the JDBC URL referring to database
- **user**—Specifies the database username
- **password**—Specifies the database password
- **var**—Specifies the name of the variable to which the DataSource object has to be set
- **scope**—Specifies the scope of the variable

Note that if dataSource attribute is used then driver, url, user and password attributes are not applicable to use.

The <sql:transaction> Tag

JSTL allows you to use transactions through the <sql:transaction> tag. You use this tag to specify a data source that you can use with the transaction. The <sql:transaction> tag groups <sql:update> and <sql:query> in its body part into a transaction.

The following is the syntax of the <sql:transaction> tag of JSTL:

```
<sql:transaction attributes>
  [<sql:query> |
  <sql:update>]+
</sql:transaction>
```

The attributes of the <sql:transaction> tag are:

- **dataSource**—Specifies the datasource JNDI name or java.sql.DataSource object to be used with this transaction.
- **isolation**—Specifies the possible values for isolation attribute are READ_COMMITTED, READ_UNCOMMITTED, REPEATABLE_READ, or SERIALIZABLE

Note that the <sql:query> or <sql:update> tags are used in <sql:transaction> tag are not allowed to have dataSource attribute.

Using JSTL SQL Tags

The jstlex3 example demonstrates the use of JSTL SQL tags, which are used to create a connection with the database. We have created datasource by using <sql:setDataSource> tag and set the datasource object in the variable "myds". Listing 12.40 is used to create the GetEmpDetails.jsp page to retrieve the employee detail from the database table. To get the data from the database table (emp), we use <sql:query> tag and execute SQL select statement whose result has been set to the variable "result". Next, the page displays the records stored in the "result" variable using the `<c:forEach>` tag. In corresponding to the each row retrieved from the database, a hyperlink is created to remove the employee from the database. When, you click on that link, a new Jsp page "RemoveEmp.jsp" opens. Listing 12.40 shows the code for the GetEmpDetails.jsp page (you can find the GetEmpDetails.jsp file in the code\Java EE\Chapter 12\jstlex4 folder on the CD):

Listing 12.40: The GetEmpDetails.jsp File

```
<%@taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%>
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<sql:setDataSource driver="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/sample"
user="root"
password="root" var="myds" scope="request"/>
<sql:query sql="select * from emp" var="result" scope="page"
  dataSource="${requestScope.myds}"/>
<html> <body>
  <table border="1">
        <tr>
        <c:forEach items="${pageScope.result.columnNames}" var="colname">
              <th><c:out value="${colname}"/></th>
        </c:forEach>
        <th> </th>
        </tr>

        <c:forEach items="${pageScope.result.rows}" var="row">
              <tr> <td>
              <c:out value="${row.empno}"/>
              </td> <td>
              <c:out value="${row.deptno}"/>
              </td> <td>
              <c:out value="${row.ename}"/>
              </td> <td>
              <c:out value="${row.sal}"/>
              </td> <td>
              <a href= "RemoveEmp.jsp?empno=<c:out value="${row.empno}"/>">
              Remove</a> </td> </tr>
        </c:forEach>
  </table>
</body> </html>
```

Listing 12.41 is used to create the RemoveEmp.jsp page to delete the employee detail from the database table "emp". To delete the employee detail from the database table, we use the <sql:query> tag and execute the DELETE SQL statement. The result of the DELETE statement is stored in the variable "count". Then, the <sql:param> tag has been used to set the "empno" in the SQL delete statement. Next, the code to check the value of the count has been written. If the value of count is equal to 1, the details corresponding to the "empno" are deleted. If the value of count is not equal to 1, the message `Problem in removing Employee` is displayed. After removing the details of the employee from the database, a link (View Employees) to navigate to the previous page has been created. Listing 12.41 shows the code for the RemoveEmp.jsp file (you can find the RemoveEmp.jsp file in the code\Java EE\Chapter 12\jstlex4 folder on the CD):

Listing 12.41: The RemoveEmp.jsp File

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%>
<sql:setDataSource driver="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/sample"
user="root"
password="root" var="myds" scope="request"/>
<sql:update dataSource="${requestScope.myds}" sql="delete from emp where empno=?"
  var="count">
<sql:param value="${param.empno}"/>
</sql:update>
<c:if test="${count eq 1}">
  <b>Employee Removed</b>
</c:if>
<c:if test="${count ne 1}">
```

```
    <b>Problem in removing Employee</b>
  </c:if>
  <br/>
  <a href="GetEmpDetails.jsp">View Employees</a>
```

Listing 12.42 shows the web.xml file in which, we have set the "GetEmpDetails.jsp" as welcome page (page that will be the starting page). Listing 12.42 shows the mapping of the GetEmpDetails.jsp page (you can find the web.xml file in the code\Java EE\Chapter 12\jstlex4\WEB-INF folder on the CD):

Listing 12.42: The web.xml File

```
<web-app>
  <welcome-file-list>
        <welcome-file> GetEmpDetails.jsp </welcome-file>
  </welcome-file-list>
</web-app>
```

After creating all the preceding files, arrange them along with jstl.jar and standard.jar as shown in Figure 12.32:

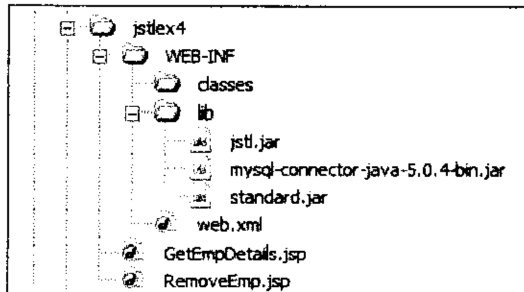

Figure 12.32: Showing Directory Structure of the jstlex4 Application

Now, copy the jstlex4 folder into the webapps folder of the Tomcat installation and start the Tomcat server. Enter the URL http://localhost:9090/jstlex4/GetEmpDetails.jsp in your browser. It displays all the employee's details as shown in Figure 12.33:

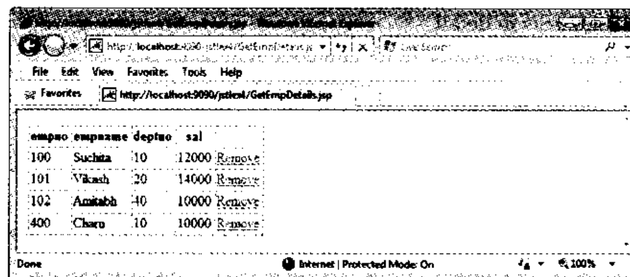

Figure 12.33: Showing Employee's Detail with Remove Option

Click on any of the Remove link to remove the respective employee; for example, clicking on the Remove link in the second row will delete the employee record with empno 101. This operation presents the view as shown in Figure 12.34:

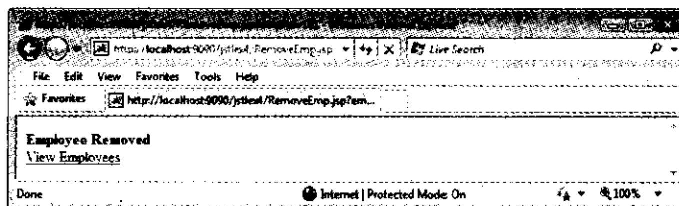

Figure 12.34: Showing Employee Removed Message

In the next section, we discuss about JSTL formatting tags.

JSTL Formatting Tags

The format Tag library provides the support for internationalization. This provides the formatting of data in different domains. The data can be dates, numbers, or the time specifications in different domains. The format tag library can be used in a JSP page by using the following specification:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

The JSTL formatting tags are divided into four categories as listed here:

- Basic Formatting Tags
- Number Formatting Tags
- Date Formatting Tags
- Time Zone Tags

Now, we discuss these categories in detail in the next sections.

Basic Formatting Tags

The JSTL basic formatting tags are used to format the data of a JSP page. These tags parse the data based on the current locale and provide support for internationalization. The <fmt:setLocale> tag sets the localization settings that are used by the <fmt:message> tag to do resource bundle look ups.

Types of Basic Formatting Tags

The tags used for formatting are:

- <fmt:setLocale>
- <fmt:setBundle>
- <fmt:bundle>
- <fmt:message>
- <fmt:param>
- <fmt:requestEncoding>

Now let's discuss each of these tags in detail.

The <fmt:setLocale> Tag

The <fmt:setLocale> tag stores the given locale in the locale configuration variable of the given scope.

The following is the syntax of the <fmt:setLocale> tag of JSTL:

```
<fmt:setLocale attributes/>
```

The attributes of the <fmt:setLocale> tag are:

- **value**—Specifies the locale, which contains a String value that contain a two-letter language code (in lower-case as defined by ISO-639), and may contain a two-letter country code (in upper-case as defined by ISO-3166). Language and country codes must be separated by hyphen '-' or underscore '_'.
- **variant**—Specifies the locale variant of the language referenced by value attribute.
- **scope**—The scope into which this object has to be set

The <fmt:setBundle> Tag

The <fmt:setBundle> tag creates a ResourceBundle object using the Locale object in the locale configuration variable and the given basename, and stores the ResourceBundle object into the given variable and scope.

The following is syntax of the <fmt:setBundle> tag of JSTL:

```
<fmt:setBundle attributes/>
```

The attributes of the <fmt:setBundle> tag are:

- **basename**—Specifies the resource bundle name
- **var**—Specifies the variable name to which this resource bundle object has to be set

- **scope**—Specifies the scope of the variable defined in "var" attribute.

The <fmt:bundle> Tag

The <fmt:bundle> tag creates a ResourceBundle object by using the Locale object in the locale configuration variable and the given basename and applies it to the formatting actions in its body content.

The following snippet shows the syntax of the <fmt:Bundle> tag of JSTL:

```
<fmt:bundle attributes> body content </fmt:bundle>
```

The attributes of <fmt:bundle> tag are listed here:

- **basename**—Specifies the resource bundle name
- **prefix**—Specifies prefix that has to be used for the messages used in this elements body content

The <fmt:message> Tag

The <fmt:message> tag maps key to localized message and performs parameter replacements using the resource bundle specified in bundle attribute.

The following is the syntax of the <fmt:message> tag of JSTL:

```
<fmt:message attributes> body content </fmt:message>
```

The attributes of <fmt:message> tag are listed here:

- **key**—Specifies the key that whose value has to be retrieved
- **bundle**—Specifies the resource bundle that has to be used to get the value of the specified key
- **var**—Specifies the variable to which the retrieved message has to be stored
- **scope**—Specifies the scope of the variable defined in "var" attribute.

The body part of <fmt:message> tag allows to use <fmt:param> tag to supply parametric values.

The <fmt:param> Tag

The <fmt:param> tag is used within the <fmt:message> element. This tag supplies the argument for the parametric replacement in a message.

The following is the syntax of the <fmt:param> tag of JSTL:

```
<fmt:param attributes/>
```

The <fmt:param> tag accepts one attribute, value, which specifies the value of the parameter to be passed.

The <fmt:requestEncoding> Tags

The <fmt:requestEncoding> tag allows to set the character encoding of a request. This tag invokes the setCharacterEncoding() method of the ServletRequest interface to set the character encoding of a request.

The following is the syntax of the <fmt:requestEncoding> tag of JSTL:

```
<fmt:requestEncoding attributes/>
```

The <fmt:requestEncoding> tag accepts one attribute, value, which represents the character encoding to be set. The character encoding is further used to decode the request parameters.

Using Basic JSTL Formatting Tags

The jstlex5 example demonstrates the usage of the basic JSTL formatting tags. Listing 12.43 shows the code to implement the concept of internationalization. The code for the I18NExample.jsp page is shown in Listing 12.43 (you can find the I18NExample.jsp file in the code\Java EE\Chapter 12\jstlex5 folder on the CD):

Listing 12.43: The I18NExample.jsp File

```
<%@taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
This example demonstrates the basic JSTL formatting tags:
<br/><br/>
Locale from client :
<b><c:out value="${pageContext.request.locale}"/></b> <br/>
<fmt:setBundle basename="ApplicationResources" var="mybundle"/>
<fmt:message key="welcome.message" bundle="${mybundle}">
```

```
        <fmt:param value="${param.uname}"/>
    </fmt:message> <br/></br>
    <b>Now testing &lt;fmt:setLocale&gt; tag:</b><br/>
    <br/><br/>
    Creating a ResourceBundle with client locale and setting it to <i>mybundle1</i>
      variable.<br/>
    <fmt:setBundle basename="ApplicationResources" var="mybundle1"/>
    Setting the locale to <i>it</i> (italian). <br/>
    <fmt:setLocale value="it"/>
    Creating a ResourceBundle with <i>it</i> (italian) locale and setting it to
      <i>mybundle2</i> variable.
    <br/><br/>
    <fmt:setBundle basename="ApplicationResources" var="mybundle2"/>
    <b>Message using <i>mybundle1</i>:</b> <br/>
      <pre>
            <fmt:message bundle="${mybundle1}" key="welcome.message">
                    <fmt:param value="${param.uname}"/>
            </fmt:message>
      </pre>
    <br/>
    <b>Message using <i>mybundle2</i>:</b> <br/>
    <pre>
      <fmt:message bundle="${mybundle2}" key="welcome.message">
            <fmt:param value="${param.uname}"/>
      </fmt:message>
    </pre>
```

Listing 12.44 shows the web.xml file to set the welcome page (you can find the web.xml file in the code\Java EE\Chapter 12\jstlex5\WEB-INF folder on the CD):

Listing 12.44: The web.xml File

```
<web-app>
  <welcome-file-list>
        <welcome-file> I18NExample.jsp </welcome-file>
  </welcome-file-list>
<web-app/>
```

Listing 12.45, 12.46, and 12.47 show the code to create properties files to set the language according to the locale. These properties files contain a key `welcome.message` and a `value` corresponding to this key. You can also find these listings in the code\Java EE\Chapter 12\jstlex5\WEB-INF\classes folder on the CD.

Listing: 12.45: ApplicationResources_en.properties

```
welcome.message=Welcome to internationalization <b>{0}</b> (English user)
```

Listing: 12.46: ApplicationResources_en_US.properties

```
welcome.message=Welcome to internationalization <b>{0}</b> (US English user)
```

Listing: 12.47: ApplicationResources_it.properties

```
welcome.message=Welcome to internationalization <b>{0}</b> (Italian user)
```

Arrange all the files in the directory structure, as shown in Figure 12.35:

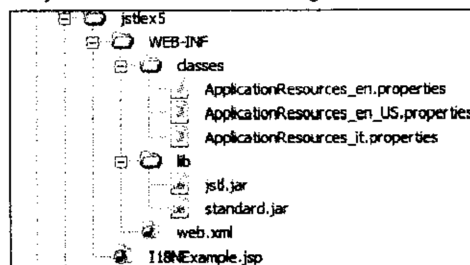

Figure 12.35: Showing directory structure of jstlex5 application

Now, copy the jstlex5 folder into webapps folder of the Tomcat installation and start the server. Browse the jstlex5 application by using the URL http://localhost:9090/jstlex5I18NExample.jsp?uname=Kogent, which presents the output as shown in Figure 12.36:

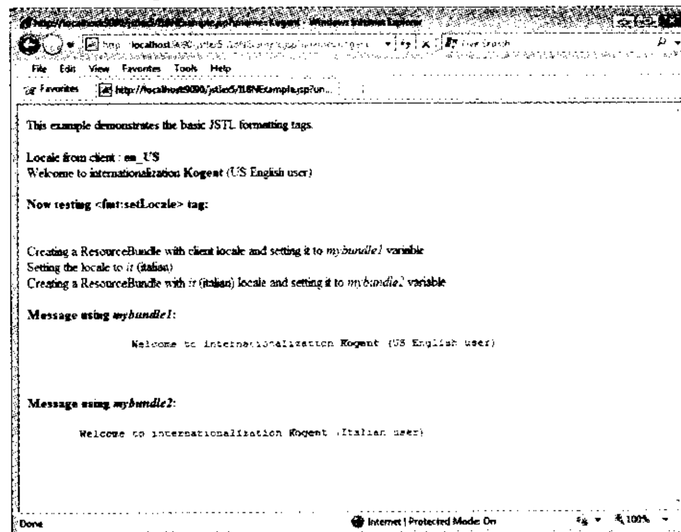

Figure 12.36: Showing Basic JSTL Formatting Tags

Number Formatting Tags

The JSTL number formatting tags are used to format the number data. The formatting aspect of number formatting tags includes the currency-related formatting, number parsing and formatting, and formatting percentages.

Types of Number Formatting Tags

The tags used for number formatting are:

- <fmt:formatNumber>
- <fmt:parseNumber>

Now let's discuss each of these tags in detail

The <fmt:formatNumber> Tag

The <fmt:formatNumber> tag allows us to format numbers, currencies, and percentages according to the locale or customized formatting pattern.

The following is the syntax of the <fmt:formatNumber> tag of JSTL:

```
<fmt:formatNumber attributes> [body content] </fmt:formatNumber>
```

The attributes of the <fmt:formatNumber> tag are:

- **value**—Specifies the numeric value that has to be formatted to the locale or given pattern. This attribute is optional and if not specified then the numeric value to be formatted should be given into the body content of this tag.
- **type**—Specifies the value type, accepted values are number, currency, and percent. If this attribute is not specified, default is taken as number.
- **pattern**—Specifies custom pattern to which the given value has to be formatted.
- **currencyCode**—Specifies the currency code as per ISO 4217 that has to be used in formatting. This is applicable only when the type is currency (i.e. when formatting currency) otherwise this attribute is ignored.

- **currencySymbol**—Specifies the currency symbol that has to be included into the formatted number. This attribute is applicable only if `type=currency` is set otherwise this attribute is ignored.
- **groupingUsed**—Specified whether the formatted output should contain any grouping separators. Takes boolean value, default is true.
- **maxIntegerDigits**—Specifies maximum number of digits in the integer portion of the formatted output.
- **minIntegerDigits**—Specifies minimum number of digits in the integer portion of the formatted output.
- **maxFractionDigits**—Specifies maximum number of digits in the fractional portion of the formatted output.
- **minFractionDigits**—Specifies minimum number of digits in the fractional portion of the formatted output.
- **var**—Specifies the variable name to which the formatted value has to be stored, if this attribute is not specified then the formatted value is written to the current JspWriter.
- **scope**—Specifies the scope of the variable defined in "var" attribute.

The <fmt:parseNumber> Tag

The <fmt:parseNumber> tag allows us to parse the string representations of numbers, currencies, and percentages formatted according to the locale or customized formatting pattern.

The following is the syntax of the <fmt:parseNumber> tag of JSTL:

```
<fmt:parseNumber attributes> [body content] </fmt:parseNumber>
```

The attributes of the <fmt:parseNumber> tag are:

- **value**—Specifies the string value that has to be parsed according to the locale or given pattern. This attribute is optional and if not specified then the string value to be parsed should be given into the body content of this tag.
- **type**—Specifies the value type, accepted values are number, currency, and percent. If this attribute is not specified, default is taken as number.
- **pattern**—Specifies the custom formatting pattern that determines how the given value is to be parsed.
- **parseLocale**—Specifies the Locale whose default formatting pattern is to be used during the parse operation, or to which the pattern specified via the pattern attribute (if present) is applied.
- **integerOnly**—Specifies whether just the integer portion of the given value should be parsed. Default is false.
- **var**—Specifies the variable name to which the parsed value has to be stored, if this attribute is not specified then the formatted value is written to the current JspWriter.
- **scope**—Specifies the scope of the variable defined in "var" attribute.

Using Number Format Tags

The jstlex6 example demonstrates you how to use JSTL number format tags. The example uses <fmt:formatNumber and <fmt:parseNumber tag.

Listing 12.48 shows the code to create index.html page to get the number from a user. This listing contains one text box, where user will enter the number, and a button. On clicking the button, the control will transfer to the TestApp.jsp (Listing 12.49). Listing 12.48 provides the code for the index.html file (you can find the index.html file in the code\Java EE\Chapter 12\jstlex6 folder on the CD):

Listing 12.48: index.html

```
<html> <body>
<form method=post action="TestApp.jsp"> <pre> <b>
  Number : <input type="text" name="mynumber"/>
    <input type="submit" value="Format Number"/>
</b> </pre> </form>
</body> </html>
```

Listing 12.49 shows the code to convert the number retrieved through index.html page into a specific format (you can find the TestApp.jsp file in the code\Java EE\Chapter 12\jstlex6 folder on the CD):

Listing 12.49: The TestApp.jsp File

```
<%@taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<html> <body>
My Number:
<b><c:out value="${param.mynumber}"/></b> </br>
Formatting My Number: <br/><br/>
<fmt:setLocale value="it"/>
In Italy:
<pre>
  Default pattern:<b>
  <fmt:formatNumber type="currency" value="${param.mynumber}"/></b>
  Using pattern (0,00,00.0000): <b>
<fmt:formatNumber type="currency" value="${param.mynumber}" pattern="00,0,00.0000"/> </b>
</pre>
<br/>
In US:
<fmt:setLocale value="en_US"/>
<pre>
  Default pattern:<b>
  <fmt:formatNumber type="currency" value="${param.mynumber}"/></b>
  Using pattern (0,00,00.0000):<b>
<fmt:formatNumber type="currency" value="${param.mynumber}" currencySymbol="$"
  pattern="0,00,00.0000" var="fmt_mynumber"/>
  <c:out value="${fmt_mynumber}"/></b>
</pre>
<br/>
After parsing the formatted mynumber: <fmt:parseNumber value="${fmt_mynumber}"
  type="currency" pattern="0,00,00.0000"/>
</body></html>
```

Listing 12.50 shows the web.xml page to set the welcome page (you can find the web.xml file in the code\Java EE\Chapter 12\jstlex6\WEB-INF folder on the CD):

Listing 12.50: The web.xml File

```
<web-app>

  <welcome-file-list>
        <welcome-file> index.html </welcome-file>
  </welcome-file-list>

</web-app>
```

Arrange all the files as shown in Figure 12.37, copy jstlex6 folder into <tomcat home>\webapps folder, and start the server.

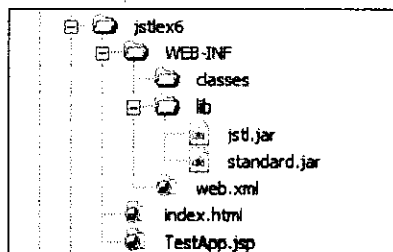

Figure 12.37: Showing Directory Structure of the jstlex6 Application

Browse the example by using the URL, `http://localhost:9090/jstlex6/index.html`, which displays a form as shown in Figure 12.38:

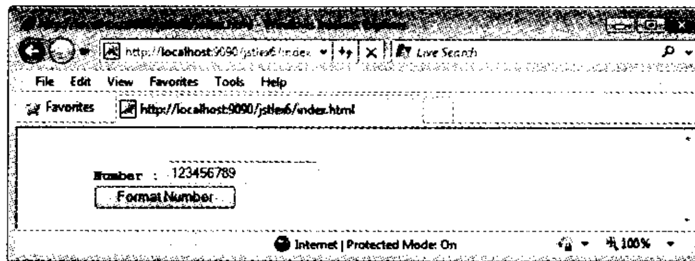

Figure 12.38: Showing the index.html Page to Enter the Number

Enter some number as shown in Figure 12.38 and click on Format Number button. This makes a request to TestApp.jsp. The TestApp.jsp formats the given number in Italian and US locale formats with default and given pattern as shown in Figure 12.39:

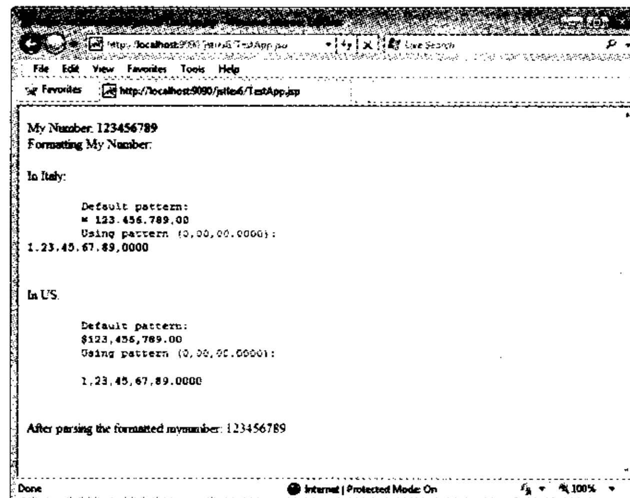

Figure 12.39: Showing JSTL Number Format Tags

Date Formatting Tags

The JSTL date formatting tags are used to format the date type of data. These tags help in formatting and parsing date-time data. The <fmt:formatDate> tag formats the date-time data and <fmt:parseDate> tag parses the date-time data.

Types of Date Formatting Tags

The tags used for date formatting are:

- <fmt:formatDate>
- <fmt:parseDate>

Now let's discuss each of these tags in detail.

The <fmt:formatDate> Tag

The <fmt:formatDate> tag allows us to format dates and times according to the locale or customized formatting pattern.

The following is the syntax of the <fmt:formatDate> tag of JSTL:

```
<fmt:formatDate attributes/>
```

The attributes of the <fmt:formatDate> tag are:

- **value**—Specifies the java.util.Date object whose Date and / or Time to be formatted.

- **type**—Specifies the components of the given date object which has to be formatted, accepted values are date, time, and both. If this attribute is not specified, default is taken as date.
- **pattern**—Specifies the custom pattern to which the given value has to be formatted.
- **dateStyle**—Specifies the predefined formatting style for dates. Accepted values are default, short, medium, long, and full. This is applicable only when the type is date or both date and time; otherwise this attribute is ignored.
- **timeStyle**—Specifies the predefined formatting style for time. Accepted values are default, short, medium, long, and full. If this attribute is not specified default is taken as default. This is applicable only when the type is time or both date and time; otherwise, this attribute is ignored.
- **timeZone**—Specifies a String value that may be one of the time zone IDs supported by the Java platform or a custom time zone ID, in which to represent the formatted time.
- **var**—Specifies the variable name to which the formatted value has to be stored, if this attribute is not specified then the formatted value is written to the current JspWriter.
- **scope**—Specifies the scope of the variable defined in "var" attribute.

The <fmt:parseDate> Tag

The <fmt:parseDate> tag allows us to parse and format the string representation of dates and times according to the locale or customized formatting pattern.

The following is the syntax of the <fmt:parseDate> tag of JSTL:

```
<fmt:parseDate attributes/>
```

The attributes of the <fmt:parseDate> tag are:

- **value**—Specifies the Date string to be parsed.
- **type**—Specifies whether the given value contains date or time or both. If this attribute is not specified default is taken as date.
- **pattern**—Specifies the custom pattern to which the given value has to be formatted.
- **dateStyle**—Specifies the predefined formatting style for dates. Accepted values are default, short, medium, long, and full. If this attribute is not specified default is taken as default. This is applicable only when the type is date or both otherwise this attribute is ignored.
- **timeStyle**—Specifies the predefined formatting style for times. Accepted values are default, short, medium, long, and full. If this attribute is not specified default is taken as default. This is applicable only when the type is time or both, otherwise this attribute is ignored.
- **timeZone**—Specifies a String value that may be one of the time zone IDs supported by the Java platform or a custom time zone ID, in which it represents the formatted time.
- **parseLocale**—Specifies the Locale whose default formatting pattern is to be used during the parse operation, or to which the pattern specified via the pattern attribute (if present) is applied.
- **var**—Specifies the variable name to which the formatted value has to be stored, if this attribute is not specified then the formatted value is written to the current JspWriter.
- **scope**—Specifies the scope of the variable defined in "var" attribute.

Using Date Formatting Tags

The Listing 12.51 of jstlex7 application demonstrates you how to use JSTL date format tags. The application uses <fmt:formatDate> and <fmt:parseDate> tag for formatting and parsing of Date information.

Listing 12.51 shows the code to format the date according to the specific locale corresponding to the specific pattern (you can find the TestApp.jsp file in the code\Java EE\Chapter 12\jstlex7 folder on the CD):

Listing 12.51: The TestApp.jsp File

```
<%@taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>

<html>
```

```
<body>
<%pageContext.setAttribute("mydate",new java.util.Date(),PageContext.PAGE_SCOPE);%>
Current Date and Time:
<b><c:out value="${mydate}"/></b> </br>
Formatting Date: <br/><br/>
<fmt:setLocale value="it"/>
In Italy:
<pre>
  Default pattern:<b>
  <fmt:formatDate type="both" value="${mydate}"/></b>
  Using pattern (dd-MMM-yyyy hh:mm:ss): <b>
<fmt:formatDate type="both" value="${mydate}" pattern="dd-MMM-yyyy hh:mm:ss"/> </b>
</pre>
<br/>
In US:
<fmt:setLocale value="en_US"/>
<pre>
  Default pattern:<b>
  <fmt:formatDate type="both" value="${mydate}"/></b>
  Using pattern (dd-MMM-yyyy hh:mm:ss): <b>
<fmt:formatDate type="both" value="${mydate}" pattern="dd-MMM-yyyy hh:mm:ss"
  var="fmt_mydate"/> </b>
  <c:out value="${fmt_mydate}"/></b>
</pre>
<br/>
After parsing the formatted mydate: <fmt:parseDate value="${fmt_mydate}" type="both"
  pattern="dd-MMM-yyyy hh:mm:ss"/>

</body>

</html>
```

Listing 12.52 shows the web.xml page to set the welcome page (you can find the web.xml file in the code\Java EE\Chapter 12\jstlex7\WEB-INF\ folder on the CD):

Listing 12.52: The web.xml File

```
<web-app>
  <welcome-file-list>
       <welcome-file>TestApp.jsp </welcome-file>
  </welcome-file-list>
</web-app>
```

Arrange all the files as shown in Figure 12.40, copy jstlex7 folder into webapps folder of the Tomcat installation and start the server. Figure 12.40 shows the directory structure of the jstlex7 application:

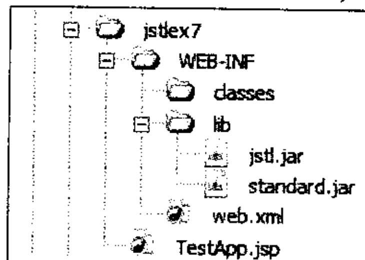

Figure 12.40: Showing Directory Structure of the jstlex7 Application

Browse the example using the URL: `http://localhost:9090/jstlex7/TestApp.jsp`, which formats current date into Italian and US locale formats with default and given pattern as shown in Figure 12.41:

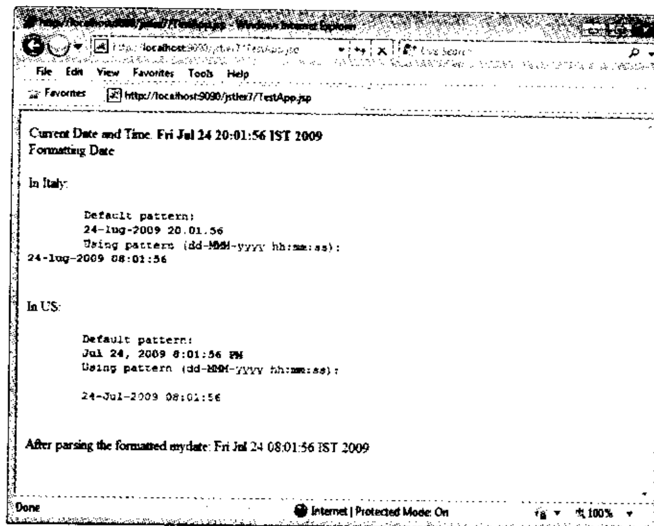

Figure 12.41: Showing JSTL Date Format Tags

Time Zone Formatting Tags

The JSTL time zone formatting tags are used to format the time zone type of data. These tags help in setting the time zone by using <fmt:setTimeZone> tag. The <fmt:timeZone> tag specifies the time zone to be used by all the tags within this tag.

Types of Time Zone Formatting Tags

The tags used for time zone formatting are:

- <fmt:timeZone>
- <fmt:setTimeZone>

Now let's discuss each of these tags in detail

The <fmt: timeZone> Tag

The body content of the <fmt: timeZone> tag specifies the tags that will be controlled by the time zone specified in <fmt:timeZone> tag.

The following is the syntax of the <fmt:timeZone> tag of JSTL:

```
<fmt:timeZone attributes> body content </fmt:timeZone>
```

The <fmt:timeZone> tag accepts one attribute, value, which specified the `java.util.TimeZone` object or a `String` that represents a time zone ID.

The <fmt:setTimeZone> Tag

The <fmt:setTimeZone> tag stores the given time zone in the time zone configuration variable of the given scope.

The following is the syntax of the <fmt:setTimeZone> tag of JSTL:

```
<fmt:setTimeZone attributes/>
```

The attributes of the <fmt:setTimeZone> tag are:

- **value**—Specifies a String value that may be one of the time zone IDs supported by the Java platform or a custom time zone ID.
- **var**—Specifies the variable name
- **scope**—Specifies the Scope into which this object has to be set

Using TimeZone Formatting Tags

The Listing 12.53 of jstlex8 example demonstrates you how to use JSTL timezone format tags. The following example uses <fmt: timeZone> tag.

Listing 12.53 shows the code to specify the time zone corresponding to a specific timezone ID (you can find the TestApp.jsp file in the in the code\Java EE\Chapter 12\jstlex8 folder on the CD):

Listing 12.53: The TestApp.jsp File

```
<%@taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<html> <body>
<%pageContext.setAttribute("mydate",new java.util.Date(),PageContext.PAGE_SCOPE);%>
Current Date and Time:
<b><c:out value="${mydate}"/></b> </br>
Formatting Date based on Time Zone: <br/><br/>
GMT:
<pre>
  <b><fmt:timeZone value="GMT">
        <fmt:formatDate type="both" value="${mydate}"/>
  </fmt:timeZone></b>
</pre>
<br/>
6 hours 30 minutes ahead of GMT :
<pre>
  <b><fmt:timeZone value="GMT+06:30">
        <fmt:formatDate type="both" value="${mydate}"/>
  </fmt:timeZone></b>
</pre>
<br/>
10 hours 30 minutes behind the GMT :
<pre>
  <b><fmt:timeZone value="GMT-10:30">
        <fmt:formatDate type="both" value="${mydate}"/>
  </fmt:timeZone></b>
</pre>
</body></html>
```

Listing 12.54 shows the web.xml file to set the welcome page (you can find the web.xml in the code\Java EE\Chapter 12\jstlex8\WEB-INF folder on the CD):

Listing 12.54: The web.xml File

```
<web-app>
  <welcome-file-list>
        <welcome-file>TestApp.jsp </welcome-file>
  </welcome-file-list>
</web-app>
```

Arrange all the files as shown in Figure 12.42, copy jstlex8 folder into webapps folder of the Tomcat installation, and start the server. Figure 12.42 shows the directory structure of the jstlex8 application:

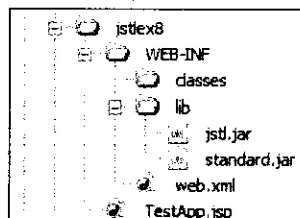

Figure 12.42: Showing Directory Structure of the jstlex8 Application

Browse the application by using the URL: `http://localhost:9090/jstlex8/TestApp.jsp`, which converts the current time of IST to GMT, GMT +06:30, and GMT -10:30 as shown in Figure 12.43:

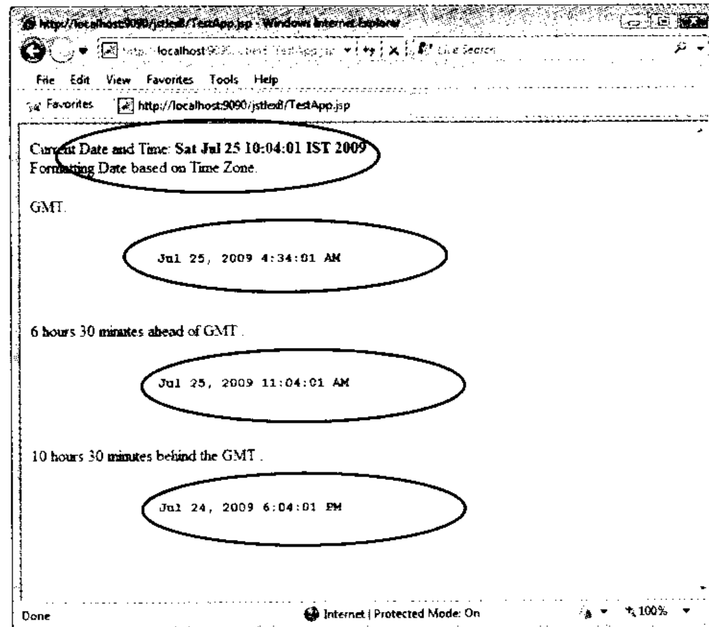

Figure 12.43: Showing Date Formatting Based on Time Zone

JSTL XML Tags

The XML tag library is used to work with XML data used in the JSP pages. The XML tag library helps parse and transform the data used in the JSP page. These operations can be done by using the XPath expressions associated with an XML document. An XPath expression includes both the relative and the absolute path of the XML element. The XML tags can be used with the JSP pages after importing the following tag library by using the following code snippet:

```
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
```

XML tags can be accessed by using the prefix 'x'. It is the predefined prefix used for XML tag libraries.

All the JSTL XML Tags can be categorized into following categories:

- XML Core Tags
- XML Flow Control Tags
- XML Transformation Tags

Let's explore all these categories in detail.

XML Core Tags

The JSTL XML Core tags include the tags for parsing XML documents and performing XPath operations. The XML Core tags are of three types:

- <x:parse>
- <x:out>
- <x:set>

The following section explains all these tags in detail.

The <x:parse> Tag

This tag parses the given XML document and stores the resulted XML Document Object Model (DOM) into the specified variable.

The following is the syntax of the <x:parse> tag of JSTL:

```
<x:parse attributes> [body content] </x:parse>
```

The attributes of the <x:parse> tag are:

- **doc**—Accepts an XML document to be parsed in the form of a string or Reader object that locates the XML document to be parsed.
- **systemId**—Accepts the system identifier (URI) for parsing the XML document.
- **filter**—Accepts an org.xml.sax.XMLFilter object to be applied for the XML document.
- **var**—Specifies the variable name to which the parsed XML document has to be set.
- **scope**—Specifies the scope of the var attribute.
- **varDom**—Specifies the variable name to which the org.w3c.dom.Document object of the parsed XML document has to be set.
- **scopeDom**—Specifies the scope of the varDom attribute.

The <x:out> Tag

The <x:out> tag evaluates an XML XPath expression and writes the result of the evaluation to the current JspWriter.

The following is the syntax of the <x:out> tag of JSTL:

```
<x:out attributes/>
```

The attributes of the <x:out> tag are:

- **select**—Accepts an XML XPath expression to be evaluated.
- **escapeXML**—Accepts a boolean value that describes whether the characters such as >, <, &, ',' in the result string has to be converted to their corresponding character entity codes. The default value is true.

The <x:set> Tag

The <x:set> tag evaluates an XML XPath expression and stores the result of the evaluation in a scoped variable.

The following is the syntax of the <x:set> tag of JSTL:

```
<x:set attributes/>
```

The attributes of the <x:out> tag are:

- **select**—Accepts an XML XPath expression to be evaluated.
- **var**—Specifies the name of the scoped variable to hold the result obtained after evaluating the given XPath expression.
- **scope**—Specifies the scope of the var attribute.

XML Flow Control Tags

The JSTL XML Flow Control tags help perform various operations such as easily parse and access XML data, iterate over elements in an XML document, and conditionally process JSP code fragments depending on the result of the given XPath expression.

The XML Flow Control tags of JSTL contain five tags:

- <x:if>
- <x:choose>
- <x:when>
- <x:otherwise>
- <x:forEach>

The following sections explain all these tags in detail.

The <x:if> Tag

The <x:if> tag evaluates an XPath expression that is given in its 'select' attribute. If the resulting value is true, then the body of the <x:if> tag is evaluated; otherwise not.

The following is the syntax of the <x:if> tag of JSTL:

```
<x:if attributes> body content </x:if>
```

The attributes of the <x:if> tag are:

- **select**—Takes an XPath expression that resolves to a Boolean value
- **var**—Takes the name of the scoped variable to store the conditions result
- **scope**—Specifies the scope of a variable

The <x:choose> Tag

The <x:choose> tag acts similar to the Java switch statement. The <x: choose> tag encloses one or more <x: when> tags and a <x:otherwise> tag.

The following is the syntax of the <x:choose> tag of JSTL:

```
<x:choose> body content </x:choose>
```

Body content of <x:choose> tag includes one or more <x:when> tags and zero or one <x:otherwise> tag.

The <x:when> Tag

The <x:when> tag encloses a single case within the <x:choose> tag.

The following is the syntax of the <x:when> tag of JSTL:

```
<x:when attribute> body content </x:when>
```

The <x:when> tag accepts only one attribute Select, which takes an XPath expression resolving to boolean value. If the Boolean value is true, the <c:when> tags body is evaluated.

The <x:otherwise> Tag

This is same as <x:when> tag but is unconditional; this is equivalent to default case in the Java switch statement. The body content of this tag is evaluated if none of the <x:when> conditions in the <x:choose> tag are resolved to true.

The following is the syntax of the <x:otherwise> tag of JSTL:

```
<x:otherwise> body content </x:otherwise>
```

The <x:forEach> tag

The <x:forEach> tag is used for looping over a list of elements obtained after evaluating the given XPath expression.

The following is syntax of the <x:forEach> tag of JSTL:

```
<x:forEach attributes> body content </x:forEach>
```

The attributes of the <x:forEach> are:

- **select**—Takes an XPath expression that results a node list
- **var**—Specifies the name of the variable into which the current iteration item has to be set
- **varStatus**—Specifies the name of the variable that lets us to know the information about where are we in the overall iteration such as getting count, index, knowing isFirst, isLast etc, this attribute is optional
- **begin**—Takes the starting index for the loop, this attribute is optional
- **end**—Takes the ending index for the loop, this attribute is optional
- **step**—Specifies an optional increment for the loop, default is 1

XML Transformation Tags

The XML Transformation tags of the JSTL XML tags provide support to transform the XML document with the given XSL stylesheet. This part of JSTL tags contain two tags, <x:transform> and <x:param>. The following section explains these tags in detail.

The <x:transform> Tag

The <x:transform> tag transforms an XML document using the given XSL style sheet.

The following is the syntax of the <x:transform> tag of JSTL:

```
<x:transform attributes> [body content] </x:transform>
```

The attributes of the <x:transform> tag are:

- **doc**—Takes an XML document to be parsed in the form of a string or a Reader object. The Reader object locates the XML document, or an org.w3c.dom.Document object or an object that is exported by <x:parse> or <x:set> tag.
- **xslt**—Takes an XSLT stylesheet document for transformation in the form of a string or a Reader object. The Reader object locates the XML document or an org.w3c.dom.Document object or an javax.xml.transform.Source object.
- **docSystemId**—Takes the system identifier (URI) for parsing the XML document.
- **xsltSystemId**—Takes the system identifier (URI) for parsing the XSLT stylesheet document.
- **var**—Takes the variable name to which the transformed XML document has to be set.
- **scope**—Specifies the scope of the var attribute.
- **result**—The javax.xml.transform.Result object that captures or processes the transformation result.

The <x:param> Tag

The <x:param> tag is used to set transformation parameters. The <x:param> tag should be used within the body of the <x:transform> tag.

The following is the syntax of the <x:param> tag of JSTL:

```
<x:param attributes> [body content] </x:param>
```

The attributes of the <x:param> tag are:

- **name**—Takes the name of the transformation parameter.
- **value**—Specifies the value of the transformation parameter. This attribute is optional; you can specify the value of the param tag in the body content if you do not want to use the value attribute.

Implementing JSTL Tags

JSTL can be used with the Web applications to access the specified tag libraries. The user needs to specify the tag libraries in the JSP page. The JSP engine is capable of accessing the specified tag library by using the Jar files available in the lib directory of the application. The tags from the specified tag library can be accessed by using the prefix of the tag library. The jstlex9 application demonstrates the use of JSTL tags in the JSP page. In Listing 12.55, the JSP pages are used to show the use of the JSTL tags in generating a Web page (you can find the GetEmpDetails.jsp file in the code\Java EE\Chapter 12\jstlex9 folder on the CD):

Listing 12.55: The GetEmpDetails.jsp File

```
<%@taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%>
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
<c:if test="${!empty param.language}" var="lang_flag">
  <fmt:setLocale value="${param.language}"/>
</c:if>
<c:if test="${!empty param.country && lang_flag}">
  <fmt:setLocale value="${param.language}_${param.country}"/>
</c:if>
<sql:setDataSource driver="com.mysql.jdbc.Driver" url="
  jdbc:mysql://localhost:3306/sample" user="root" password="root" var="myds"
  scope="request"/>
<sql:query sql="select empno,deptno,ename,sal,hiredate from employee" var="result"
  scope="page" dataSource="${requestScope.myds}"/>
<html> <body>
  Display Currency and Date in: <pre>
  <a href="GetEmpDetails.jsp?language=en&country=US">English(US)</a> <a
  href="GetEmpDetails.jsp?language=en&country=gb">English(UK)</a> <a
```

```
href="GetEmpDetails.jsp?language=en&country=AU">English(AU)</a> <a
href="GetEmpDetails.jsp?language=it">Italian</a>
</pre>
<table border="1">
<tr>
        <c:forEach items="${pageScope.result.columnNames}" var="colname">
        <th><c:out value="${colname}"/></th>
        </c:forEach>
</tr>
<c:forEach items="${pageScope.result.rows}" var="row">
<tr> <td>
        <c:out value="${row.empno}"/>
</td> <td>
        <c:out value="${row.deptno}"/>
</td> <td>
        <c:out value="${row.ename}"/>
</td> <td>
        <fmt:formatNumber value="${row.sal}" type="currency"/>
</td> <td>
        <fmt:formatDate value="${row.hiredate}" type="date"/>
</td> </tr>
</c:forEach>
</table>
</body> </html>
```

Listing 12.56 shows the web.xml file to set the welcome page (you can find the web.xml in the code\Java EE\Chapter 12\jstlex9\WEB-INF folder):

Listing 12.56: The web.xml File

```
<web-app>
  <welcome-file-list>
        <welcome-file> GetEmpDetails.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Figure 12.44 shows the tree structure of the jstlex9 application:

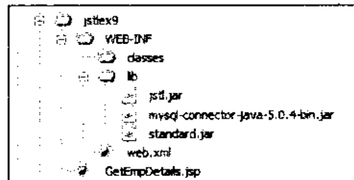

Figure 12.44: Showing Directory Structure of the jstlex9 Application

Copy jstlex9 folder into <tomcat home>\webapps folder, and start the server. Browsing GetEmpDetails.jsp page with the URL `http://localhost:9090/jstlex9/GetEmpDetails.jsp` which presents the employee details as shown in Figure 12.45, considering the browser language which is configured to en-US:

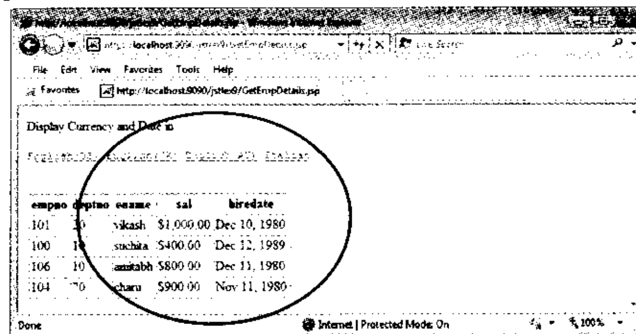

Figure 12.45: Displaying Currency and Sate in US English

Now, to view the date and salary in the default format of UK, click the English (UK) hyperlink in Figure 12.45. The default format of salary (currency) and data in UK are displayed as shown in Figure 12.46:

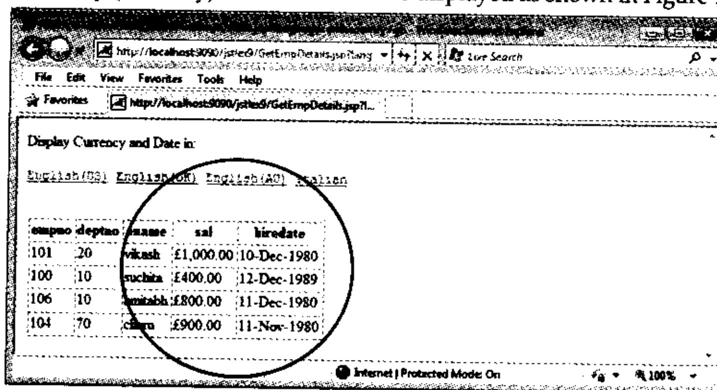

Figure 12.46: Displaying Currency and Date in UK English

To view the date and salary in the default format of Australia, click the English (AU) hyperlink in Figure 12.46. The default format of salary (currency) and data in Australia are displayed as shown in Figure 12.47:

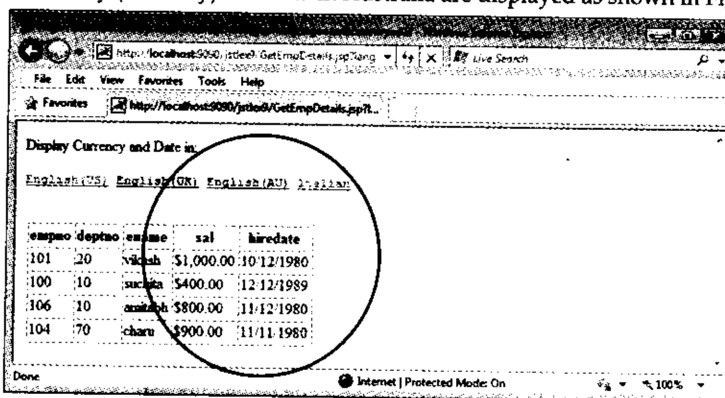

Figure 12.47: Displaying Currency and date in English (AU)

To view the date and salary in the default format of Italy, click the Italian hyperlink in Figure 12.47. The default format of salary (currency) and data in Italy are displayed as shown in Figure 12.48:

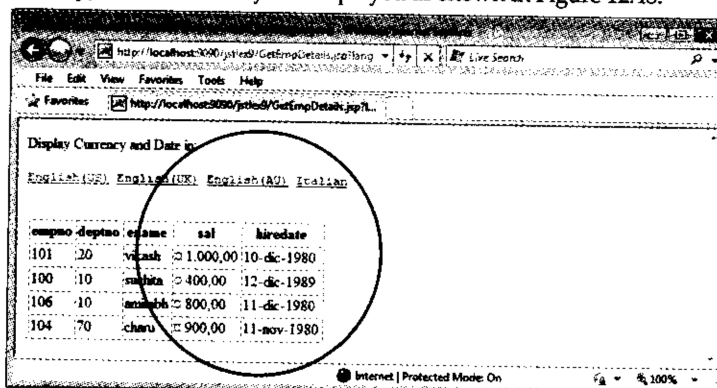

Figure 12.48: Displaying Currency and Date in Italian

In this application, we have used the core, SQL, and formatting tags to create a connection with database and to format the data to be displayed on JSP page. The JSTL tag library is imported in the JSP page to implement these JSTL tags.

Summary

This chapter described Java Server Pages (JSP), a technology used to create dynamic Web pages. After introducing JSP, the chapter covered the advantages of JSP over Servlets to create complex views. It also described JSP architecture and then given a detailed description of the various stages of the JSP life-cycle.

In this chapter, you have also learned about various types of JSP scripting tags, implicit objects, and JSP directives. Scripting tags allow us to write the Java code in JSP pages. You also learned how to create JSP pages by using the scripting tags. You learned that implicit objects are used in a JSP page to access the Java code. JSP directives are used to import a Java file into a JSP page to control the processing of the entire page in an application. JSP directives allow you to include other pages into the context of a JSP page. The JSP action tags and bean components available in a JSP page are also discussed in this chapter. You have learned, action tags such as <jsp:useBean>, <jsp:setProperty>, <jsp:getProperty>, and <jsp:text> along with the syntax to use them. Then, we created an application that helps in creating, declaring, and reading the properties of a bean. We also deployed and executed the application to see its output.

Towards the end, the chapter has familiarized you with the JSTL core tags, SQL tags, formatting tags, and XML tags. This chapter has also demonstrated the implementation of JSTL tags with the help of numerous applications. The next chapter discusses the concepts of JDBC.

Quick Revise

Q.1 What is JSP?

Ans. JSP stands for Java Server Pages. JSP is a standard Java extension used to simplify the creation and management of dynamic Web pages.

Q.2 What are the stages of the JSP lifecycle?

Ans.
- Page Translation
- Compilation
- Loading & Initialization
- Request Handling
- Destroying (End of service)

Q.3 Why is JSP preferred over Servlets?

Ans. Both, Servlets and JSPs are server-side components used to generate dynamic HTML pages; however, JSP is preferred over Servlet as it is developed by using a simple HTML template and is automatically handled by the JSP container.

Q.4 Name the scripting tags available in JSP.

Ans. Scriptlet, Declarative, and Expression

Q.5 Which of the following are implicit objects?

Option 1: request

Option 2: config

Option 3: context

Option 4: session

Ans. Options 1, 2, and 4

Q.6 What is the type of implicit object exception?

Ans. java.lang.Throwable

Q.7 Name the directive tags available as pre JSP specifications.

Ans.
- page
- include
- taglib

Q.8 What is the use of page and include directive tags?

Ans.
- **Page**—The page directive tag holds the instructions used by the translator during the translation stage of the JSP lifecycle. These instructions affect the various properties associated with the JSP page.
- **Include**—The include directive tag is used to merge the contents of two or more files during the translation stage of the JSP lifecycle.

Q.9 Name the attributes of the <jsp:useBean> tag.

Ans.
- id
- scope
- class
- beanName
- type

Q.10 What is the use of fallback action tag?

Ans. The fallback action tag allows us to specify a text message to be displayed if the required plug-in cannot run.

Q.11 What is JSTL?

Ans. JSP Standard Tag Library (JSTL) is a collection of custom tag libraries, which provide core functionality used for JSP documents.

Q.12 How many types of tag libraries are provided by JSTL?

Ans. JSTL provides 4 types of tag libraries that can be used with JSP pages.

Q.13 What are JSTL core tags used for?

Ans. The JSTL core tags are used to perform iteration, conditional processing, and also provide support of expression language for the tags in JSP pages.

Q.14 What is the function of the <c:import> tag?

Ans. The <c: import> tag is used to include another resource, such as another JSP or HTML page, in a JSP page.

Q.15 What is the difference between <c:import> and <jsp:include>?

Ans. The <c: import> tag is similar to the <jsp: include> tag. However, the <c: import> tag allows us to include the pages in another Web application. This tag also allows us to save the included page output into a variable instead of directly writing it to the output, which allows us to process the included page output before writing it to the output, if necessary.

Q.16 Why are JSTL SQL tags used?

Ans. The SQL tags are used to access the relational database specified in JSP pages. The SQL tags are used for rapid prototyping and developing Web applications.

Q.17 What are JSTL formatting tags used for?

Ans. The format Tag library provides the support for internationalization. This tag provides the formatting of data in different domains. The data can be dates, numbers, or time specifications in different domains.

Q.18 What are JSTL XML tags used for?

Ans. The XML tag library is used to work with XML data used in the JSP pages. The XML tag library helps parse and transform the data used in a JSP page.