TABLE OF CONTENTS (SHOW)

Java Server-Side Programming

JavaServer Pages (JSP) (1.2 and 2.0)

1. Introduction

Instead of *static* contents that are indifferent, *Java Servlet* was introduced to generate *dynamic* web contents that are customized according to users' requests (e.g., in response to queries and search requests). However, it is a pain to use a Servlet to produce a presentable HTML page (via the `out.prinltn()` programming statements). It is even worse to maintain or modify that HTML page produced. Programmers, who wrote the servlet, may not be a good graphic designer, while a graphic designer does not understand Java programming.

JavaServer Pages (JSP) is a *complimentary* technology to *Java Servlet* which facilitates the *mixing* of dynamic and static web contents. JSP is Java's answer to the popular Microsoft's *Active Server Pages* (ASP). JSP, like ASP, provides a elegant way to mix static and dynamic contents. The main page is written in regular HTML, while special tags are provided to insert pieces of Java programming codes. The *business programming logic* and the *presentation* are cleanly separated. This allows the programmers to focus on the business logic, while the web designer to concentrate on the presentation.

JSP is based on Servlet. In fact, we shall see later that a JSP page is internally translated into a Java servlet. We shall also explain later that "***Servlet is HTML inside Java***", while "***JSP is Java inside HTML***". Whatever you can't do in servlet, you can't do in JSP. JSP makes the creation and maintenance of dynamic HTML pages much easier than servlet. JSP is more convenience than servlet for dealing with the presentation, not more powerful.

JSP is meant to *compliment* Servlet, not a replacement. In a *Model-View-Control* (MVC) design, servlets are used for the controller, which involves complex programming logic. JSPs are used for the view, which deals with presentation. The model could be implemented using JavaBeans or Enterprise JavaBeans (EJB) which may interface with a database.

To understand JSP, you need to understand Servlet (and HTTP, and HTML, and Java). Read "Java Servlet", if necessary.

The JSP Home Page is @ http://www.oracle.com/technetwork/java/javaee/jsp/index.html. For developers, check out JSP Developer Site @ http://jsp.java.net.

Advantages of JSP

- *Separation of static and dynamic contents*: The dynamic contents are generated via programming logic and inserted into the *static template*. This greatly simplifies the creation and maintenance of web contents.
- *Reuse of components and tag libraries*: The dynamic contents can be provided by re-usable components such as JavaBean, Enterprise JavaBean (EJB) and tag libraries - you do not have to re-inventing the wheels.
- Java's power and portability.

Other Technologies

[TODO] Competing technologies: ASP, PHP, etc.

[TODO] Complementary and extension technologies: MVC Framework (JSF, Struts, Spring, Hibernate), AJAX.

Apache Tomcat Server

JSPs, like servlets, are server-side programs run inside a *Java-capable* HTTP server. Apache Tomcat Server (@ http://tomcat.apache.org) is the official reference implementation (RI) for Java servlet and JSP, provided free by Apache (@ http://www.apache.org) - an open-source software foundation.

You need to install Tomcat to try out JSP. Read "How to Install Tomcat and Get Started", if necessary.

I shall denote Tomcat's installed directory as `<CATALINA_HOME>`, and assume that Tomcat server is running in port 8080.

Tomcat provides many excellent JSP examples, in "`<CATALINA_HOME>\webapps\examples\jsp`". You can run these examples by launching Tomcat, and issue URL `http://localhost:8080/examples`.

JSP Versions

JSP has these versions: [TODO features and what is new]

- J2EE 1.2 (December 12, 1999) (Java Servlet 2.2, **JSP 1.1**, EJB 1.1, JDBC 2.0)
- J2EE 1.3 (September 24, 2001) (Java Servlet 2.3, **JSP 1.2**, EJB 2.0, JDBC 2.1)
- J2EE 1.4 (November 11, 2003) (Java Servlet 2.4, **JSP 2.0**, EJB 2.1, JDBC 3.0)
- Java EE 5 (May 11, 2006) (Java Servlet 2.5, **JSP 2.1**, JSTL 1.2, JSF 1.2, EJB 3.0, JDBC 3.0)
- Java EE 6 (December 10, 2009) (Java Servlet 3.0, **JSP 2.2/EL 2.2**, JSTL 1.2, JSF 2.0, EJB 3.1, JDBC 4.0)
- Java EE 7: expected in end of 2012

2. First JSP Example - "Java inside HTML"

Let's begin with a simple JSP example.

First of all, create a new *web application* (aka *web context*) called "`hellojsp`" in Tomcat, by creating a directory "`hellojsp`" under Tomcat's "webapps" directory (i.e., "`<CATALINA_HOME>\webapps\hellojsp`" where `<CATALINA_HOME>` denotes Tomcat's installed directory). The "`<CATALINA_HOME>\webapps\hellojsp`" directory is known as *context root* for webapp (web context) "`hellojsp`".

Use a programming text editor to enter the following HTML/JSP codes and save as "`first.jsp`" under the context root "`hellojsp`". The file type of "`.jsp`" is mandatory for JSP script.

```
2  <%@page language="java" contentType="text/html" pageEncoding="UTF-8" %>
3  <!DOCTYPE html>
4  <html>
5  <head>
```

```
6     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7     <title>First JSP</title>
8  </head>
9
10 <body>
11    <%
12    double num = Math.random();
13    if (num > 0.95) {
14    %>
15       <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
16    <%
17    } else {
18    %>
19       <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
20    <%
21    }
22    %>
23    <a href="<%= request.getRequestURI() %>"><h3>Try Again</h3></a>
24 </body>
25 </html>
```

To run this JSP script, launch the Tomcat server. Check the console message to confirm that `hellojsp` has been deployed:

```
xxxx, xxxx xx:xx:xx xx org.apache.catalina.startup.HostConfig deployDirectory
INFO: Deploying web application directory hellojsp
......
```

Start a browser. Issue this URL (assume that Tomcat is running in port number 8080):

```
http://localhost:8080/hellojsp/first.jsp
```

Well, life goes on ...

(0.16450910536318764)

Try Aagin

Try "View Source" to check the *response message* received by the browser.

```
<!DOCTYPE HTML>
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
   <title>First JSP</title>
</head>
<body>
   <h2>Well, life goes on ... </h2><p>(0.16450910536318764)</p>
   <a href="/hellojsp/first.jsp"><h3>Try Again</h3></a>
</body>
</html>
```

Observe that the response message does not include the JSP source codes, but merely the *output* of the JSP script. This clearly illustrates that JSP (like servlets) are server-side programs, that are executed in the server. The output is then sent to the client (browser) as the response message.

Explanations

- A JSP script is a *regular HTML page* containing additional Java codes.
- In line 1, we declare this is a JSP script via a JSP directive. The `<!DOCTYPE>` tag identifies that the response is an HTML document. The `<html>...</html>` tags enclose the HTML document, which consists of two sections: head and body, enclosed by `<head>...</head>` and `<body>...</body>`, respectively.
- The JSP's Java codes are enclosed within special tags in the form of `<% ... %>` (similar to ASP). We shall explain these codes later.
- "***JSP is Java inside HTML***" - Java codes are embedded inside an HTML page. On the other hand, "***Servlet is HTML inside Java***" - HTML page is produced using Java's `out.println()` in a Java program.

2.1 Revisit Java Servlets

A typical Java servlet (as shown below) contains three groups of methods: `init()`, `destroy()`, and one or more `service()` methods such as `doGet()` and `doPost()`. `init()` runs (once) when the servlet is loaded into the server. `destroy()` runs (once) when the servlet is unloaded. `service()` runs once per HTTP request (e.g., `doGet()` runs once per GET request, `doPost()` run once per POST request). The `service()` methods takes two arguments: `request` and `response`, encapsulating HTTP request and response messages respectively. A `PrintWriter` object called `out` is used for writing out the response message to the client over the network.

```
public class MyServlet extends HttpServlet {
   // Instance variables or methods, e.g., database Connection, Statement, helper methods
   ......
   ......

   // init() runs only once when the servlet is loaded into the server
   public void init() { ...... }

   // doGet() runs once per HTTP GET request
   // It takes two arguments, representing the request and response messages
   public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws IOException, ServletException {

      // Set the MIME type for the response message
      response.setContentType("text/html");
      // Create a Writer to write the response message to the client over the network
      PrintWriter out = response.getWriter();

      // The programming logic to produce a HTML page
      out.println("<html>");
      out.println( ..... );
```

```
      out.println("</html>");
   }

   // doPost() runs once per HTTP Post request
   public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws IOException, ServletException {
      ......
   }

   // destroy() runs only once when the servlet is unloaded from the server.
   public void destroy() { ..... }
}
```

2.2 Behind the Scene

When a JSP page is first requested, Tomcat translates the JSP into a servlet, compiles the servlet, load, and execute the servlet. The best way to understand JSP is to check out the generated servlet and study the JSP-to-Servlet translation. The generated servlet for "`first.jsp`" is kept under Tomcat's "`work`" directory ("`<CATALINA_HOME>\work\Catalina\localhost\hellojsp\...`"). The relevant part of the generated servlet is extracted below:

```
public final class first_jsp extends org.apache.jasper.runtime.HttpJspBase
      implements org.apache.jasper.runtime.JspSourceDependent {

   public void _jspInit() { ..... }

   public void _jspDestroy() { .....  }

   public void _jspService(HttpServletRequest request, HttpServletResponse response)
         throws java.io.IOException, ServletException {

      // JSP pre-defined variables (in service() method)
      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 {
         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("<!DOCTYPE HTML>\r\n");
         out.write("<html>\r\n");
         out.write("<head>\r\n");
         out.write("   <meta http-equiv=\"Content-Type\" content=\"text/html; charset=UTF-8\">");
         out.write("   <title>First JSP</title>\r\n");
         out.write("</head>\r\n");
         out.write("\r\n");
         out.write("<body>\r\n");
         out.write("   ");

         double num = Math.random();
         if (num > 0.95) {
            out.write("\r\n");
            out.write("      <h2>You will have a luck day!</h2><p>(");
            out.print( num );
            out.write(")</p>\r\n");
            out.write("   ");
         } else {
            out.write("\r\n");
            out.write("      <h2>Well, life goes on ... </h2><p>(");
            out.print( num );
            out.write(")</p>\r\n");
            out.write("   ");
         }
         out.write("\r\n");
         out.write("   <a href=\"");
         out.print( request.getRequestURI() );
         out.write("\"><h3>Try Again</h3></a>\r\n");
         out.write("</body>\r\n");
         out.write("</html>\r\n");
      } catch (Throwable t) {
         if (!(t instanceof SkipPageException)) {
            out = _jspx_out;
            if (out != null && out.getBufferSize() != 0)
               try { out.clearBuffer(); } catch (java.io.IOException e) {}
            if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
         }
      } finally {
         _jspxFactory.releasePageContext(_jspx_page_context);
      }
   }
}
```

Clearly, the translation is carried out as follows:

- The methods `_jspInit()`, `_jspDestroy()` and `_jspService()` corresponds to `init()`, `destroy()` and `service()` (`doGet()`, `doPost()`) of a regular servlet. Similar to servlet's

- The methods `_jspInit()`, `_jspDestroy()` and `_jspService()` corresponds to `init()`, `destroy()` and `service()` (`doGet()`, `doPost()`) of a regular servlet. Similar to servlet's `service()`, `_jspService()` takes two parameters, `request` and `response`, encapsulating the HTTP request and response messages. A `JspWriter` called `out`, corresponding to servlet's `PrintWriter`, is allocated to write the response message over the network to the client.
- The HTML statements in the JSP script are written out as part of the response via `out.write(...)`, as "it is" without modification.
- The Java codes in the JSP script are translated according to their respective types:
  - *JSP Scriptlet* `<% .... >`: used to include Java statements, or part of Java statement. The Java statements are placed inside the `_jspService()` method of the translated servlet as "it is". Scriptlets form the program logic.
  - *JSP Expression* `<%= .... >`: used to *evaluate* a single Java expression to obtain a value. The Java expression is placed inside a `out.print(...)`. In other words, the Java expression will be evaluated, and the result of the evaluation written out as part of the response message.

Subsequent accesses to the this JSP page will be much faster, because they will be re-directed to the translated and compiled servlet directly (no JSP-to-servlet translation and servlet compilation needed again), unless the JSP page has been modified.

2.3 JSP Pre-Defined Variables

JSP pre-defined seven variables, that are available to the script writer. They are: `request`, `response`, `out`, `session`, `application`, `config`, and page.

`request`: A `HttpServletRequest` object, keeping track of the HTTP request message. It is often used to retrieve the query parameters in the request message. For example,

```
// Single-value parameter
String paramValue = request.getParameter("paramName");
if (paramValue != null && paramValue.legnth() != 0) {  // param exists and not empty string
   // process the parameter
   ......
}

// Multiple-value parameter
String[] paramValues = request.getParameterValues("paramName");
if (paramValues != null && paramValues.length > 0) {  // param exists and at least one item
   // Process parameters
   for (String paramValue : paramValues) {
      ......
   }
}
```

`response`: A `HttpServletResponse` object, keeping track of the HTTP response message.

`out`: A `Writer` (`JspWriter`) object used to write response message to the client over the network socket, via methods `print()` or `println()`.

`session`: A `HttpSession` object, keeping track of the current client session (from the moment the user accesses the first page, until he/she closes the browser or session timeout). You can use `session`'s attributes to pass information between pages within this session, via methods `getAttribute("name")` and `setAttribute("name", object)`. For example,

```
// Allocate a shopping cart
List<String> shoppingCart = new ArrayList<>();
.....
// Place the shopping cart inside the session
session.setAttribute("cart", shoppingCart);
.....
// Any page can retrieve the shopping cart
List<String> theCart = (List<String>)session.getAttribute("cart");
if (theCart != null) {  // cart exists?
   for (String item : theCart) {  // process the cart items
      ......
   }
}
```

`application`: A `ServletContext` object retrieved via `getServletContext()`, which maintains information about this web context (web application). You can use the `application`'s attributes to pass information between JSP pages and servlets, via methods `getAttribute("name")` and `setAttribute("name", object)`.

`config`: A `ServletConfig` object, obtained via `getServletConfig()`. It could be used to retrieve the servlet initialization parameters provided in "`WEB-INF\web.xml`", via method `getInitParameter("paramName")`.

`page`: Can be used to access the elements of this page.

2.4 Behind the Scene

The translated servlet shows these seven variables are allocated and initialized as follows (with re-arrangements):

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
   javax.servlet.jsp.PageContext pageContext
         = _jspxFactory.getPageContext(this, request, response, null, true, 8192, true);
         /* getPageContext(javax.servlet.Servlet servlet,
                           javax.servlet.ServletRequest request,
                           javax.servlet.ServletResponse response,
                           java.lang.String errorPageURL,
                           boolean needsSession,
                           int bufferSize,
                           boolean autoFlushBuffer)  */

   javax.servlet.http.HttpSession session = pageContext.getSession();
   javax.servlet.ServletContext application = pageContext.getServletContext();
   javax.servlet.ServletConfig config = pageContext.getServletConfig();
   javax.servlet.jsp.JspWriter out = pageContext.getOut();
   java.lang.Object page = this;
   ......
```

2.5 Brief Summary

Compare the JSP script and the internally generated servlet, we can clearly see that servlet is "HTML inside Java", whereas JSP is "Java inside HTML". In a servlet, you need to use `out.write()` to write out the HTML page, which is difficult to write, inflexible to changes, hard for graphic designers (non-programmers), and mixing the presentation (view), data (model), with the business logic (control). In JSP, the main page is an HTML page, meant for presentation (to be done by graphic designer). Pieces of Java codes (written by programmer) are embedded into the HTML file to perform the business logic. The presentation and business logic can be cleanly separated.

3. Another JSP Example

Write the following JSP script and save as "`echo.jsp`" in your web application's root directory. This example displays a form with checkboxes, and echos your selections.

```
<%@page language="java" contentType="text/html" pageEncoding="UTF-8" %>
<!DOCTYPE HTML >
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Echoing HTML Request Parameters</title>
</head>

<body>
  <h2>Choose authors:</h2>
  <form method="get">
    <input type="checkbox" name="author" value="Tan Ah Teck">Tan
    <input type="checkbox" name="author" value="Mohd Ali">Ali
    <input type="checkbox" name="author" value="Kumar">Kumar
    <input type="checkbox" name="author" value="Peter Johnson">Peter
    <input type="submit" value="Query">
  </form>

  <%
  String[] authors = request.getParameterValues("author");
  if (authors != null) {
  %>
    <h3>You have selected author(s):</h3>
    <ul>
      <%
      for (String author : authors) {
      %>
        <li><%= author %></li>
      <%
      }
      %>
    </ul>
  <%
  }
  %>
  <br /><a href="<%= request.getRequestURI() %>">BACK</a>
<body>
</html>
```

Run the JSP page and study the generated servlet.

Explanation:

- This page has a HTML form with 4 checkboxes, identified by their "`name=value`" pairs of "`author=xxx`". No "`action`" attribute is specified in the `<form>` tag. The default "`action`" is the current page (i.e. the query will be sent to the same page for processing).
- The JSP scriptlet checks if the query parameter "`author`" exists. For the first request, "`author`" parameter is absent. Once the user fills in and submits the form, "`author`" will be present in the HTTP request.
- `request.getParameterValues()` is used to retrieve all the values of the query parameter "`author`". The values are echoed back to the client in an unordered list.

4. JSP Scripting Elements

JSP scripting element are enclosed within `<% ..... %>`, similar to other server-side scripts such as ASP and PHP. To print "<%", use escape sequence "<\%".

JSP Comment `<%-- comments --%>`

JSP comments `<%-- JSP comments --%>` are ignored by the JSP engine. For example,

```
<%-- anything but a closing tag here will be ignored -->
```

Note that HTML comment is `<!-- html comments -->`. JSP expression within the HTML comment will be evaluated. For example,

```
<!-- HTML comments here <%= Math.random() %> more comments -->
```

JSP Expression `<%= JavaExpression %>`

A JSP expression is used to insert the resultant value of a single Java expression into the response message. The Java expression will be placed inside a `out.print(...)` method. Hence, the expression will be evaluated and resultant value printed out as part of the response message. Any valid Java expression can be used. There is no semi-colon at the end of the expression.

For examples:

```
<%= Math.sqrt(5) %>
<%= item[10] %>
<p>The current data and time is: <%= new java.util.Date() %></p>
```

The above JSP expressions will be converted to:

```
out.print( Math.sqrt(5) );
out.print( item[10] );
out.write("<p>Current time is: ");
out.print( new java.util.Date() );
out.write("</p>");
```

You can use the pre-defined variables in the expressions. For examples:

```
<p>You have choose author <%= request.getParameter("author") %></p>
<%= request.getRequestURI() %>
```

You can also use the XML-compatible syntax of `<jsp:expression>Java Expression</jsp:expression>`.

JSP Scriptlet <% Java Statements %>

JSP scriptlets allow you to implement more complex programming logic. You can use scriptlets to insert any valid Java statements into the `_jspService()` method of the translated servlet. The Java codes must be syntactically correct, with Java statements terminated by a semi-colon.

For example:

```
<%
String author = request.getParameter("author");
if (author != null && !author.equals(""))) {
%>
   <p>You have choose author <%= author %></p>
<%
}
%>
```

In the translated servlet, the above will be inserted into the `service()` method as follows:

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
   ......
   String author = request.getParameter("author");
   if (author != null && !author.equals(""))) {
      out.write("<p>You have choose author ");
      out.print(author);
      out.write("</p>");
   }
}
```

The source codes of scriptlets are only available in the server, and not sent to the client. That is, scriptlets are safe and secure!

You can also use the XML-compatible syntax of `<jsp:scriptlet>Java Statements</jsp:scriptlet>`.

JSP Declaration <%! Java Statements %>

JSP declarations can be used to define variables and methods for the class. Unlike scriptlets that are inserted inside the `_jspService()` method, the declaration codes are inserted inside the class, at the same level as `_jspService()`, as variables or methods of the class.

For example,

```
<%! private int count; %>
<%! public int incrementCount() { ++count; } %>
```

will be translated to:

```
public final class first_jsp extends org.apache.jasper.runtime.HttpJspBase
      implements org.apache.jasper.runtime.JspSourceDependent {

   private int count;
   public int incrementCount() { ++count; }

   public void _jspInit() { ..... }

   public void _jspDestroy() { .....  }

   public void _jspService(HttpServletRequest request, HttpServletResponse response)
         throws java.io.IOException, ServletException { ..... }
}
```

You can also use the XML-compatible syntax of `<jsp:declaration>Java Statements</jsp:declaration>`.

JSP Directive <%@ ... %>

JSP directives provide instructions to the JSP engine to aid in the translation. The syntax of the JSP directive is:

```
<%@ directiveName attribute1="attribute1value" attribute2="attribute2value" ... %>
```

The directives include: `page`, `include`, `taglib`.

JSP Page Directive <%@page ... %>

Typically, a JSP file starts with the page directive:

```
<%@page language="java" contentType="text/html" %>
```

The "`page import`" directive lets you import classes. The syntax is:

```
<%@page import="packageName.*" %>
<%@page import="packageName.className" %>
<%@page import="packageName.className1, packageName.className2,..." %>
```

This directive generates proper import statements at the top of the servlet class. User-defined classes must be stored in "*webContextRoot*`\WEB-INF\classes`", and kept in a proper package (default package not allowed).

For example,

```
<%-- import package java.sql.* --%>
<%@page import="java.sql.*" %>

<%-- import a user-defined class --%>
<%@page import="mypkg.myClass1, mypkg.myClass2" %>
```

The "`page`" directives are also used to set the MIME type, character set of the response message. These information will be written to the response message's header. For example,

```
<%-- Set the response message's MIME type, character set --%>
<%@page contentType="image/gif" %>
<%@page contentType="text/html" charset="UTF-8" %>
<%@page pageEncoding="UTF-8" %>

<%-- Set an information message for getServletInfo() method --%>
```

```
<%@page info="Hello world example" %>
```

The "page session" directive allows you to designate whether this page belongs to the session. The default is "true". Setting to "false" could reduce the server's load, if session tracking is not needed in your application.

```
<%@page session="true|false" @>
```

Other "page" directives include: errorPage, isErrorPage, buffer, isThreadSafe, isELIgnored, etc.

JSP Include Directive <%@include ... %>

The "include" directive lets you insert the unprocessed content of an external file. You can include any JSP files or static HTML files. You can use include directive to include navigation bar, copyright statement, logo, etc. on every JSP pages. The syntax is:

```
<%@include file="url" %>
<%@include page="url" &>
```

If another JSP page is included using "include file", the JSP engine will not check for update of the included JSP file during execution.

For example:

```
<%@include file="header.jsp" %>
....
<%@include file="footer.jsp" %>
```

JSP Taglib Directive <%@taglib ... %>

JSP allows you to create your own libraries of JSP tags. You can use the taglib directive to tell Tomcat what libraries to load and where they are. For example,

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

Taglib will be elaborated later.

5. JSP Standard Actions

The purpose of JSP actions is to specify activities to be performed when a page is processed. In contrast, JSP scripting elements such as directives (<%@ ... %>) are processed when translating the page, which produces static contents.

A JSP action may contain sub-action. The syntax is as follows.

```
<jsp:action-name attribute-list>
   <jsp:subaction-name subaction-attribute-list />
</jsp:action-name>
```

There are eight JSP standard actions (forward, include, useBean, getProperty, setProperty, plugin, text, element) and five sub-actions (attribute, body, fallback, param, and params).

<jsp:forward>

To terminate execution of the current page and forward the request to another page. For example,

```
<jsp:forward page="nextPage.jsp">
  <jsp:param name="parmName" value="parmValue"/>
</jsp:forward>
```

The destination page can access the new parameters via request.getParameter(paramName). Tomcat clears the output buffer upon executing a forward action.

<jsp:include>

To execute another page and append its *output* to the current page. That is, it places the generated HTML code into the current JSP page. This is different from the JSP include directive <%@ include ... %>, which insert *unprocessed* content.

```
<jsp:include page="headers.jsp"/>
```

Behind the Scene

<jsp:forward> is translated as follows:

```
<jsp:forward page="index.jsp" />
// is translated to
_jspx_page_context.forward("index.jsp");
// Re-direct, or forward the current ServletRequest and ServletResponse to
//   another active component in the application.
```

<jsp:include> is translated as follows:

```
<jsp:include page="index.jsp"/>
// is translated to
org.apache.jasper.runtime.JspRuntimeLibrary.include(request, response, "index.jsp", out, false);
// Invoke the include method during execution of the servlet.
// Cause the target page to be processed, and output written to the response message.
// The last boolean argument specifies whether JspWriter is to be flushed before the include.
```

6. Using JavaBeans in JSP (JSP Actions useBean, getProperty, setProperty)

6.1 Revisit JavaBeans

A *JavaBean* is a Java class which conforms to the following rules:

- It has a no-arg constructor.
- It does not have public variables.
- It is defined in a named package. It can not be kept in the default no-name package.

- It is defined in a named package. It can not be kept in the default no-name package.
- For a private variable `xxx`, there is a public getter `getXxx()` (or `isXxx()` for `boolean`) and a public setter `setXxx()`.
- For an event `xxxEvent`, there is an interface `xxxListener`, and methods `addXxxListener()` and `removeXxxListener()`.
- It implements `Serializable` interface, so that its state can be stored and retrieved to and from external storage, for persistent.

6.2 JSP Actions for JavaBean

<jsp:useBean>

Create a new bean instance. The bean class must be kept in "`<WebContextRoot>\WEB-INF\classes`" within a proper package directory structure.

```
<jsp:useBean id="beanName" class="packageName.ClassName" scope="page|request|session|application" />
// Equivalent to JSP Declaration:
// <%! PackageName.className beanName = new PackageName.className(); %>
// setAttribute according to the scope.
```

The attribute `scope` specifies the scope of this bean:

- `page`: default, stored in `PageContext`, available to this page only.
- `request`: stored in `ServletRequest`, can be forwarded to another JSP page or servlet.
- `session`: stored in `HttpSession`, available to all pages and servlets within this session.
- `application`: stored in `ServletContext`, shared by all servlets and JSP pages in this web context (application).

<jsp:setProperty>

Set the value of a property (variable) of a bean, by invoking its setter.

```
<jsp:setProperty name="beanName" property="propertyName" value="propertyValue" />
// Equivalent to Scriptlet: <% beanName.setPropertyName(propertyValue); %>
```

<jsp:getProperty>

Get the value of a property (variable) of a bean, by invoking its getter.

```
<jsp:getProperty name="beanName" property="propertyName" />
// Equivalent to Expression: <%= beanName.getPropertyName() %>
```

6.3 JSP's JavaBean and HTML form

Bean is frequently used in a HTML form, to capture the value of the selected request parameters (such as username), and pass it over to the other processing pages or server-side programs, depending on its scope.

Instead of setting each of the bean's properties using many `<jsp:setProperty>`, you can use wildcard `'*'` in the `property` attribute to set the properties of a bean to the request parameters with the matching names. Simple type conversion from String will be carried out. For example,

```
<jsp:useBean name="mybean" class="mypkg.myClass" scope="session' />
  // Instantiate a bean
<jsp:setProperty name="mybean" property="*" />
  // Binds the properties with matching request parameter names
  // Same as mybean.setXxx(request.getParameter("xxx")) for all the properties in the bean
```

Suppose the request contains a parameter `username=xxx` and `mybean` has a property `username`, then `setUsername(xxx)` will be invoked.

6.4 A JSP with JavaBean Example

1. Let's create a JavaBean called "`UserBean.java`", which contains a variable (property) called `username`, with getter and setter as follows:

```
package mypkg;

public class UserBean {

   private String username;

   public UserBean() {
      username = "";
   }

   public String getUsername() {
      return username;
   }

   public void setUsername(String username) {
      this.username = username;
   }
}
```

2. Next, create a input form to prompt user for his/her name called `index.jsp`:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML >
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Input Form</title>
  </head>
  <body>
    <h1>Hello,</h1>
    <form action="response.jsp">
      Enter your name: <input type="text" name="username" />
      <input type="submit" />
    </form>
  </body>
</html>
```

3. The Input Form submits the request to "`response.jsp`", as below:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML >
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Response Page</title>
  </head>
  <body>
    <jsp:useBean id="user" scope="session" class="mypkg.UserBean" />
    <jsp:setProperty name="user" property="*" />
    <h1>Hello, <jsp:getProperty name="user" property="username" />!</h1>
  </body>
</html>
```

- The `<jsp:useBean>` directive construct an instance of `mypkg.UserBean` called "`user`". This bean has scope of `session`, which is kept in the HTTPSession, and available to all JSP pages and servlets within this session.
- The `<jsp:setProperty>` sets the value of `username` in `mykpg.UserBean` to the value of the request parameter "`username`" (i.e., `user.setUsername(request.getParameter("username"))`).
- The `<jsp:getProperty>` retrieves the `username` of `mypkg.UserBean` (i.e., `user.getUserName()`).

Behind the Scene

For the above example, the `<jsp:useBean>`, `<jsp:setProperty>` and `<jsp:getProperty>` for "`session`" scope is translated as follows:

```
<jsp:useBean id="user" scope="session" class="mypkg.UserBean"  />
<jsp:setProperty name="user" property="username" />
<h1>Hello, <jsp:getProperty name="user" property="username" />!</h1>
```

```
// <jsp:useBean>
mypkg.UserBean user = null;
synchronized (session) {  // prevent concurrent update
   // Check if bean already exists in the HTTPSession.
   // If so, retrieve from the session
   user = (mypkg.UserBean) _jspx_page_context.getAttribute(
         "user", javax.servlet.jsp.PageContext.SESSION_SCOPE);
   if (user == null) {
      // Otherwise, allocate an instance, and place it in the session
      user = new mypkg.UserBean();
      _jspx_page_context.setAttribute(
            "user", user, javax.servlet.jsp.PageContext.SESSION_SCOPE);
   }
}
// <jsp:setProperty>
org.apache.jasper.runtime.JspRuntimeLibrary.introspecthelper(
      _jspx_page_context.findAttribute("user"), "username",
      request.getParameter("username"), request, "username", false);
// <jsp:getProperty>
out.write(org.apache.jasper.runtime.JspRuntimeLibrary.toString(
      (((mypkg.UserBean)_jspx_page_context.findAttribute("user")).getUsername())));
```

`<jsp:setProperty>` with wildcard '`*`':

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

```
org.apache.jasper.runtime.JspRuntimeLibrary.introspect(
      _jspx_page_context.findAttribute("user"),  request);
```

`<jsp:useBean>` with "`application`" scope:

```
<jsp:useBean id="user" scope="application" class="mypkg.UserBean"  />
```

```
mypkg.UserBean user = null;
synchronized (application) {
   user = (mypkg.UserBean) _jspx_page_context.getAttribute(
         "user",  javax.servlet.jsp.PageContext.APPLICATION_SCOPE);
   if (user == null) {
      user = new mypkg.UserBean();
      _jspx_page_context.setAttribute(
         "user", user,  javax.servlet.jsp.PageContext.APPLICATION_SCOPE);
   }
}
```

6.5 Tomcat's JSP 1.2 Example - Number Guess

Tomcat provides many excellent examples on JSP and servlets (kept under "`webapps\examples`"), but lack appropriate explanation for newbies.

Let's illustrate the use of JavaBean using the Tomcat's JSP sample "number guess". I modified the codes slightly to fit my style. You can find the original JSP source at "`webapps\examples\jsp\num\numguess.jsp`", and Javabean at "`webapps\examples\WEB-INF\classes\num\NumberGuessBean.java`". You can run the "number guess" by issuing URL http://localhost:8080/examples/jsp/num/numguess.jsp.

Let's run our examples in web context "`hellojsp`" created earlier. We now need to create the proper directory structure for a web application and put the files at the right place. First create a directory "`hellojsp`" under "`webapps`". Then, create a sub-directory "`WEB-INF`" under "`hellojsp`". Finally, create sub-directories "`classes`", "`src`" and "`lib`" under "`WEB-INF`". Take note that the directory names are case-sensitive.

Put the files in the appropriate directories:

- "`<CATALINA_HOME>\webapps\hellojsp`": This directory is known as *context root* of web context "`hellojsp`". Keep the "`jsp`" and "`html`" files available to the users.
- "`<CATALINA_HOME>\webapps\hellojsp\WEB-INF`": Keep the configuration files, such as "`web.xml`".
- "`<CATALINA_HOME>\webapps\hellojsp\WEB-INF\src`": Keep the java program source files (optional).
- "`<CATALINA_HOME>\webapps\hellojsp\WEB-INF\classes`": Keep the java classes.
- "`<CATALINA_HOME>\webapps\hellojsp\WEB-INF\lib`": keep the "`jar`" files, which could be provided by external libraries.

JavaBean - "NumberGuessBean.java"

Save the source code as "<CATALINA_HOME>\webapps\hellojsp\WEB-INF\src\mypkg\NumberGuessBean.java"

```
package mypkg;            // Bean must be kept in a named package
import java.util.Random;  // For generating random number
import java.io.Serializable;

// NumberGuessBean, with session scope, for storing game data.
public class NumberGuessBean implements Serializable {

   private int answer;       // the random number to be guessed
   private boolean success;  // got the answer?
   private String hint;      // hinting messages, e.g., "higher", "lower"
   private int numGuesses;   // number of guesses made so far, init to 0
   private Random random;    // random number generator

   // Constructor
   public NumberGuessBean() {
      random = new Random(); // Init the random number generator
      reset();
   }

   // Reset the game
   public void reset() {
      // Generate a random int between 1 to 100, uniformly distributed.
      answer = random.nextInt(100) + 1;
      success = false;
      numGuesses = 0;
   }

   // <jsp:setProperty name="numguess" property="*"/> bounds to this method
   //   because setGuess() matches request parameter "guess".
   // Always set the values of numGuesses and hint.
   // Set success if user got the answer right.
   public void setGuess(String strIn) {
      ++numGuesses;  // one more guess step

      int numIn;
      // Convert the input String to an int.
      // Catch if the input string does not contain a valid int.
      try {
         numIn = Integer.parseInt(strIn);
      } catch (NumberFormatException e) {
         hint = "a number next time";
         return;
      }

      if (numIn == answer) {
         success = true;
      } else if (numIn < answer) {
         hint = "higher";
      } else if (numIn > answer) {
         hint = "lower";
      }
   }

   // Public getters for private variables success, hint and numGuesses.
   // No public setters. Values are set in the setGuess() method.
   public boolean isSuccess() { return success; }
   public String getHint() { return hint; }
   public int getNumGuesses() { return numGuesses; }
}
```

Notes:

- To compile this Java code using JDK, use `-d` option to put the compiled class into the "`hellojsp\WEB-INF\classes`" directory. The compiler automatically creates the proper package directory structure.

  ```
  -- Change directory to "<CATALINA_HOME>\webapps\hellojsp\WEB-INF\classes"
  d:> cd \<CATALINA_HOME>\webapps\hellojsp\WEB-INF\classes
  -- Specify the destination using -d option
  d:\<CATALINA_HOME>\webapps\hellojsp\WEB-INF> javac -d . ..\src\mypkg\NumberGuessBean.java
  ```

  The output `NumberGuessBean.class` will be created in "`<CATALINA_HOME>\webapps\hellojsp\WEB-INF\classes\mypkg`".
- A proper Javabean is kept in a named package, implements `Serializable`, has no public variables, has a no-arg constructor (default in this example), getters and setters.
- The central method is `setGuess()`. This method is bound to the JSP via `<jsp:setProperty name="numguess" property="*"/>`. We shall see later in the JSP script that a request parameter called "guess=xxx" is used to send the user input, which is matched to this method via the wildcard `'*'` property.
- It is interesting to see that there is no private variable `guess`, no setters for `success`, `hint`, and `numGuesses`.

JSP Script - "numguess.jsp"

Write the main page "numguess.jsp" and save under the context root "hellojsp".

```
<%@page import = "mypkg.NumberGuessBean" %>
<jsp:useBean id="numguess" class="mypkg.NumberGuessBean" scope="session" />
<%-- This JSP has a form with request parameter "guess=xxx", which matches
     the setGuess() method of the bean  --%>
<jsp:setProperty name="numguess" property="*"/>

<html>
<head><title>Number Guess</title></head>
<body>
<%-- For the first access without request parameter "guess", setGuess() did not run.
     For subsequent access with parameter "guess", setGuess() (bound earlier in jsp:setProperty)
```

```
    already run to check the user input. --%>
<%
if (numguess.isSuccess()) {    // correct guess - updated by setGuess() bound earlier
%>
  <p>Congratulations!  You got it, after <%= numguess.getNumGuesses() %> tries.</p>
  <% numguess.reset(); %>
  <p><a href="<%= request.getRequestURI() %>">try again</a></p>
<%
} else {
  if (numguess.getNumGuesses() == 0 ) {   // start a game
%>
    <p>Welcome to the Number Guess game.</p>
<%
  } else {   // in a game, wrong guess
%>
    <p>Good guess, but nope. Try <strong><%= numguess.getHint() %></strong>.</p>
    <p>You have made <%= numguess.getNumGuesses() %> guesses.</p>
<%
  }
%>
  <%-- Putting up the form to get the user guess  --%>
  <p>I'm thinking of a number between 1 and 100.<p>
  <form method="get">
    <p>What's your guess? <input type="text" name="guess">
    <input type="submit" value="Submit">
  </form>
<%
}
%>
</body>
</html>
```

Explanation

- First of all, take note that this JSP contains a HTML form (at the bottom). The <form> tag does not specify an "`action`", which is defaulted to the current page. The form contains a text field with named "`guess`".
- In Line 1, the "`page import`" JSP directive import our JavaBean. The `<jsp:useBean>` creates an instance of the bean with `scope="session"`. We need the session scope, as the game is played over multiple requests.
- The `<jsp:setProperty>` uses wildcard `'*'` for the property. The request parameter "`guess`" (defined in the form) matches the method `setGuess()` in the bean. For the first access of this page, there is no request parameter, and nothing bounds. For subsequent accesses, the user input is carried via the request parameter "`guess`", which trigger `setGuess()` to check for the user input, and update the game status accordingly.

Behind the Scene

The codes relevant to the JavaBean of the compiled servlet (in "`$CATALINA\work\hellojsp\...`") are extracted as follows:

```
import mypkg.NumberGuessBean;   // Produced by "page import" directive
.....
public final class numguess_jsp extends org.apache.jasper.runtime.HttpJspBase
      implements org.apache.jasper.runtime.JspSourceDependent {

   public void _jspInit() { ..... }
   public void _jspDestroy() { ..... }

   public void _jspService(HttpServletRequest request, HttpServletResponse response)
         throws java.io.IOException, ServletException {

      // Setting up JSP pre-defined variables
      ......

      // <jsp:useBean> produces these
      mypkg.NumberGuessBean numguess = null;
      // Scope is session. If numguess already exists, use the existing. Otherwise, allocate one.
      synchronized (session) {
         numguess = (mypkg.NumberGuessBean) _jspx_page_context.getAttribute("numguess", PageContext.SESSION_SCOPE);
         if (numguess == null){
            numguess = new mypkg.NumberGuessBean();
            _jspx_page_context.setAttribute("numguess", numguess, PageContext.SESSION_SCOPE);
         }
      }
      .......
      // <jsp:setProperty> with wildcard property
      // Match the request parameters to the bean's properties
      org.apache.jasper.runtime.JspRuntimeLibrary.introspect(
            _jspx_page_context.findAttribute("numguess"), request);
      .......
}
```

7. Init and Destroy

If you need to place codes into the `init()` and `destroy()` methods of the servlet, e.g., maintain a shared database connection, you could override the `jspInit()` and `jspDestroy()` methods.

For example, the following JSP script creates a shared database `Connection` and `Statement` for accessing a database in the `init()`, for all the requests.

```
<%@ page import = "java.sql.*" %>
<%!
private Connection conn;
private Statement stmt;

public void jspInit() {
   try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  // Not needed in JDK 6
```

```
      conn = DriverManager.getConnection("jdbc:odbc:eshopDBC");
   } catch (ClassNotFoundException ex) {
      ex.printStackTrace();
   } catch (SQLException ex) {
      ex.printStackTrace();
   }
}

public void jspDestroy() {
   try {
      conn.close();
   } catch (SQLException ex) {
      ex.printStackTrace();
   }
}
%>
```

8. Tomcat's JSP 1.2 Examples

8.1 Date

Use a bean with "page" scope to display the current date/time (using `Calendar` class), where session tracking is not required. [TODO]

8.2 Snoop

[TODO]

8.3 Error Page

[TODO]

8.4 Session Carts

[TODO]

9. JSP Custom Tag

In addition to the standard actions like <jsp:useBean>, JSP allows you to define your own custom tag to perform a custom action, to extent the functions. You can use a custom tag in your JSP page as follows:

```
<prefix:actionTag attributeName="attributeValue" />
```

To define custom JSP tag, you need the followings:

1. Create *Tag Handlers*: Define the Java classes that perform the actions, including the definition of the attributes. These classes are called Tag Handlers.
2. Provide a *Tag Library Descriptor File*: An XML file (with file type of "`.tld`") describing tag name, attributes, and the implementation tag handler class, so that Tomcat knows how to handle the custom tag. It shall be kept in "*<WebContextRoot>*`\WEB-INF`".
3. In the JSP files that uses the custom tags, you need to specify the location of tag library and define a tag prefix.

9.1 Example: HelloTag

Let's write a custom standalone tag called <eg:hello /> to say "Hello, world!".

Step 1: Create a Tag Library Descriptor (TLD) File: A TLD file contains *tag descriptor*s, which map a custom tag to a tag handler class. Create a TLD file called "`mycustomtag.tld`" under "`<ContextRoot>\WEB-INF\tlds`", as follows. Currently, this TLD file contains no tag descriptor.

(NetBeans) Right-click on "`WEB-INF`" ⇒ New ⇒ Tag Library Descriptor ⇒ In "TLD name", enter "`mycustomtag`" ⇒ In "Folder", enter "`tlds/`" (default) ⇒ In "URI", enter "`mycustomtag`" ⇒ In "Prefix", enter "eg" ⇒ Finish.

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" 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-jsptaglibrary_2_1.xsd">

  <tlib-version>1.0</tlib-version>
  <short-name>eg</short-name>
  <uri>mycustomtag</uri>

</taglib>
```

The <short-name> declares the *default* tag prefix. The *optional* <uri> define a public URI that uniquely identifies this tag library. Starting from JSP 1.2, the container (such as Tomcat) uses an auto-discovery feature to expose this URI to all the JSP pages. [We no longer need to edit the `web.xml` to deploy custom tags with the <taglib> element.]

Step 2: Write the Tag Handler Class: Create a *simple* tag handler class called "`HelloTag.java`", which extends `SimpleTagSupport`.

(NetBeans) Right-click on "Source Package" ⇒ New ⇒ Tag Handler... ⇒ In "Class Name", enter "`HelloTag`" ⇒ In "Package", enter "`mypkg`" ⇒ In "Tag Support Class to Extend", select "`SimpleTagSupport`" ⇒ Next ⇒ In TLD File, browse and select "`WEB-INF\tlds\mycustomtag.tld`" (created earlier) ⇒ In "Tag Name", enter "`hello`" ⇒ In "Body Content", select "empty".

```
package mypkg;

import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class HelloTag extends SimpleTagSupport {
   @Override
   public void doTag() throws JspException {
      JspWriter out = getJspContext().getOut();
      try {
         out.println("<h1>Hello world from custom tag!</h1>");
```

```
      } catch (java.io.IOException ex) {
         throw new JspException("Error in HelloTag ", ex);
      }
   }
}
```

The program obtains the `Writer`, and prints "Hello world".

Step 3: Write a Tag Descriptor: Next, create a tag descriptor for the `HelloTag`, by editing the "`mycustomtag.tld`" created earlier. The name of the tag is *<prefix*`:hello`>, which maps to "`mypkg.HelloTag.class`". This is a standalone tag without body content.

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" 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-jsptaglibrary_2_1.xsd">

  <tlib-version>1.0</tlib-version>
  <short-name>eg</short-name>
  <uri>mycustomtag</uri>

  <tag>
    <name>hello</name>
    <tag-class>mypkg.HelloTag</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

Step 4: Write a JSP: We can now write a JSP page (called "`testHelloTag.jsp`") to use the custom tag created.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="eg" uri="mycustomtag" %>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Test Custom Tag - Hello</title>
   </head>
   <body>
      <eg:hello />
   </body>
</html>
```

The `Taglib` directive's attribute "`uri`" locates the TLD "`mycustomtag.tld`" based on the public URI exposed. Alternatively, you can specify the TLD filename, i.e., `uri="/WEB-INF/tlds/mycustomtag.tld"`; or to a JAR file that has a TLD file at location `META-INF/taglib.tld`. It also sets the prefix of the custom tag. `<eg:hello />` invokes the tag handler.

Run the JSP: Start the Tomcat and issue a URL to select the "`testHelloTag.jsp`".

9.2 Interface SimpleTag and Class SimpleTagSupport (JSP 2.0)

A simple tag handler implements `javax.servlet.jsp.tagext.SimpleTag` interface, which declares five abstract methods:

```
void doTag()
  // To be called back by the container to invoke this tag.
JspTag getParent()
  // Returns the parent of this tag, for collaboration purposes.
void setJspBody(JspFragment jspBody)
  // To be called back by the container to provides
  //   the body of this tag as a JspFragment object.
  // You could retrieve this JspFragment via getJspBody().
void setJspContext(JspContext pc)
  // To be called back by the container to provide this tag handler
  //   with the JspContext for this invocation.
  // You can retrieve this JspContext via getJspContext().
void setParent(JspTag parent)
  // Sets the parent of this tag, for collaboration purposes.
```

The life cycle for handling simple tag invocation, e.g., `<eg:hello />`, is:

1. The container creates an instance of the tag handler, using the no-arg constructor.
2. The container calls back the `setJspContext()`; and `setParent()` if the tag is nested within any tag invocation. The earlier example uses `getJspContext().getOut()` to retrieve the `JspWriter` to write the response.
3. The container calls back the setter(s) of all the attribute(s).
4. If a body exists, the container calls back the `setJspBody()` method to set the body of the tag as a `JspFragment`.
5. The container calls back the `doTag()`, which caries out all the business logic.

Instead of implementing the `SimpleTag` interface, you could extend the adaptor class `SimpleTagSupport`, which provides default implementation to all the abstract methods, and override the desired methods (as in the earlier example).

9.3 Handling the Tag's Attributes

Let's rewrite the `HelloTag` (called `HelloAttributeTag`) to handle an attribute called "`name`". That is, `<eg:hello-attr name="Peter" />` prints "Hello, Peter!".

HelloAttributeTag.java

```
package mypkg;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class HelloAttributeTag extends SimpleTagSupport {

   private String name;

   // The container calls back this setter to process attribute "name"
```

```
   public void setName(String name) {
      this.name = name;
   }

   @Override
   public void doTag() throws JspException {
      JspWriter out = getJspContext().getOut();
      try {
         if (name != null) {
            out.println("<h1>Hello, " + name + "!</h1>");
         } else {
            out.println("<h1>Hello, everybody!</h1>");
         }
      } catch (java.io.IOException ex) {
         throw new JspException("Error in HelloAttributeTag ", ex);
      }
   }
}
```

mycustomtag.tld

Add the following tag descriptor into the "mycustomtag.tld".

```
<taglib ......>
  ......

  <tag>
    <name>hello-attr</name>
    <tag-class>mypkg.HelloAttributeTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
       <name>name</name>
       <required>false</required>
       <rtexprvalue>true</rtexprvalue>
       <type>java.lang.String</type>
    </attribute>
  </tag>
</taglib>
```

The name of this tag is <eg:hello-attr>. An optional attribute (required is false) called "name" is declared. It accepts "runtime (request-time) expression values" (rtexprvalue). In other words, we can use this tag in any of the following ways:

```
<eg:hello-attr name="Peter" />
  // via the setter setName()
<eg:hello-attr />
  // name is null, allowed as required=false
<eg:hello-attr name="${param['name']}" />
  // via the request parameter name
  // Enabled via rtexprvalue=true
  // E.g., http://localhost:8080/path/testHelloTag.jsp?name=Paul
```

testHelloTag.jsp

Modify "testHelloTag.jsp" to include these three statements:

```
<eg:hello-attr name="Peter" />
<eg:hello-attr />
<eg:hello-attr name="${param['name']}" />
```

Issue a URL with a query parameter, e.g., http://localhost:8080/path/testHelloTag.jsp**?name=Paul**. The outputs are:

```
Hello, Peter!       // from the name attribute
Hello, everybody!   // no attribute
Hello, Paul!        // evaluated on request-time from the request parameter
```

9.4 Processing the Tag's Body Content

Let's rewrite the HelloTag (called HelloBodyTag) to handle the body content. That is, <eg:hello-attr>Peter</eg:hello-attr> prints "Hello, Peter!".

HelloBodyTag.java

```
package mypkg;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HelloBodyTag extends SimpleTagSupport {

   @Override
   public void doTag() throws JspException {
      JspWriter out = getJspContext().getOut();
      try {
         out.println("<h1>Hello, ");
         JspFragment body = getJspBody();
         if (body != null) {
            // Print the body content as it is
            body.invoke(out);
         } else {
            out.println("everybody");
         }
         out.println("!</h1>");
      } catch (java.io.IOException ex) {
         throw new JspException("Error in HelloTag ", ex);
      }
   }
}
```

The container calls back `setJspBody()` (implemented by `SimpleTagSupport`), which makes the tag's body content available as an executable `JspFragment` of any EL expressions, standard actions, custom actions and template text. In the `doTag()` method, we can access this `Jspfragment` with the `getJspBody()` method, and execute it via the `invoke()` method.

mycustomtag.tld

Add the following tag descriptor into the "`mycustomtag.tld`".

```
<taglib ......>
  ......

  <tag>
    <name>hello-body</name>
    <tag-class>mypkg.HelloBodyTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
</taglib>
```

Instead of `body-content` of `empty` in the earlier cases, we set it to `scriptless`. This allows us to put EL expressions, standard actions, custom actions and template text within the tag's body; but not java code.

testHelloTag.jsp

Modify "`testHelloTag.jsp`" to include these statements, and run with request parameter, e.g., `http://localhost:8080/path/testHelloTag.jsp?name=Alvin`.

```
<eg:hello-body>Alan</eg:hello-body>              // Hello, Alan!
<eg:hello-body>${param['name']}</eg:hello-body>  // Hello, Alvin!
<eg:hello-body></eg:hello-body>                  // Hello, everybody!
<eg:hello-body />                                // Hello, everybody!
```

9.5 Brief Summary on SimpleTagSupport

Simple tag handlers (extending `SimpleTagSupport`) are easy to write and have more than enough power to get most of the things done. Although you are not permitted to use scriptlet inside the body, but you can get around with other means. Also, simple tag handler creates an instance for each occurrence in the JSP page, which might be an issue in some situations. Beside SimpleTagSupport, you could extend `BodyTagSupport`, which give you more flexibility.

9.6 Interface BodyTag and Class BodyTagSupport

The interface BodyTag and its superclasses super-interfaces `IterationTag`, `JspTag`, and `Tag`, declares these abstract methods:

```
void setBodyContent(BodyContent b)
  // Set the bodyContent property.
void doInitBody()
  // Prepare for evaluation of the body.
int doAfterBody()
  // Process body (re)evaluation.
int doStartTag()
  // Process the start tag for this instance.
int doEndTag()
  // Process the end tag for this instance.
void release()
  // Called on a Tag handler to release state.
void setPageContext(PageContext pc)
  // Set the current page context.
void setParent(Tag t)
  // Set the parent (closest enclosing tag handler) of this tag handler.
Tag getParent()
  // Get the parent (closest enclosing tag handler) for this tag handler.
```

Instead of implementing interface BodyTag, we could extend adaptor class `BodyTagSupport`, which provides default implementation to all the abstract methods, and additional methods for convenience. You only need to override the desired methods.

Example [TODO]

9.7 Custom Functions

[TODO] Based on Tomcat's JSP example.

10. JSP Standard Tag Library (JSTL) & Expression Language (EL)

10.1 Introduction to JSTL

JSP Standard Tag Library (JSTL) is *a collection of custom tags* that provides support for common task, to reduce the needs of writing scripting elements (scriptlets). It consists of five tag libraries (a) core, (b) internationalization (i18n) and formatting, (c) XML processing, (d) Database access, and (e) functions.

The mother site for JSTL is http://java.sun.com/products/jsp/jstl. You could download the JSTL API specification. There are a few JSTL versions:

- JSTL 1.0 (JSP 1.2, Servlet 2.3)
- JSTL 1.1 (JSP 2.0, Servlet 2.4, part of Java EE 1.4)
- JSTL 1.2 (JSP 2.1, Servlet 2.5, part of Java EE 5).

To use JSTL tag library in Tomcat, download the JSTL binaries from Apache's "Taglibs" Project @ http://tomcat.apache.org/taglibs/standard. Unzip and copy "`standard.jar`" and "`jstl.jar`" (under "`lib`" directory) to Tomcat's `<CATALINA_HOME>\lib` (server-wide); or `<ContextRoot>\WEB-INF\lib` (for the specific web application only). Alternatively, these two jar-files are available at `<CATALINA_HOME>\webapps\examples\WEB-INF\lib`.

Before getting into JSTL in depth, let us first look at Expression Language (EL).

10.2 Introduction to Expression Language (EL)

EL was introduced in JSP 2.0 as an alternative to the scripting elements (scriptlets), to make writing JSP easier by non-programmers and JSP pages more readable. For example, the following JSP

with scriptlets is hard to write, and equally hard to read:

```
<% String username = request.getParameter("username");
   if (username != null) { %>
      <p>Hello, <%= username %>!</p>
<% } else { %>
      <p>Hello, everyone!</p>
<% } %>
```

It can be re-written with JSTL and EL as follows:

```
<p>Hello, <c:out default="everyone" value="${param['username']}" />!</p>
```

EL is loosely based on ECmaScript (JavaScript) and the XML Path Language (XPath). EL is a *data-access* scripting language. It is geared toward looking up objects and their properties, and performing simple operations on them; it is not a programming language (or even a scripting language) as it lacks programming constructs. It is called "expression" language because its main construct `${expression}` evaluates the *expression* contained within the braces.

10.3 EL Syntax

EL is a very simple language with only a few syntactic rules.

Expression Evaluators `${...}` and `#{...}`

The main construct of the EL is the expression evaluator `${ELExpression}`, which evaluates the *ELExpression* within the braces. For example, `${1+2*3}`, `${index == 0}`, `${user.firstName}`, `${empty param.user}`, `${aBean.aProperty}`, `${aMap["aKey"]}`.

`${...}` evaluates the expression during compilation. On the other hand, the request-time evaluator `#{...}` evaluates the expression during request. More on request-time evaluator later.

Literals

Any value that does not begin with `${` is treated as a *literal*. EL's literals include:

- numbers: integer and floating point (similar to Java's syntax).
- strings: string can be enclosed with either single quotes or double quotes. Use escape sequence `\'`, `\"`, `\\` for `'`, `"`, `\` respectively.
- booleans: `true` or `false`.
- `null`.

Example [TODO]

EL's Implicit Objects

Similar to JSP, EL has these implicit objects:

- **`pageContext`**: The JSP *page* context. It provides access to other objects. For example:
  - **`pageContext.servletContext`** returns the *application* context;
  - **`pageContext.session`** returns the *session*;
  - **`pageContext.request`** returns the *request*,
  - **`pageContext.response`** returns the *response*.
- **`param`**: A key-value map of request parameters to their *first* value. E.g., `${param["username"]}` or `${param.username}`.
- **`paramValues`**: A key-value map of request parameters to *arrays* of their values. E.g., `${param["language"]}`, with multiple values.
- **`header`**, **`headerValues`**: A key-value map of request headers to their first value and to arrays of their values, respectively. E.g., `${header["host"]}`, `${header["accept"]}`, `${header["user-agent"]}`.
- **`cookie`**: A key-value map of cookies' name and value.
- **`initParam`**: A key-value map of the application's initialization parameter declared in "`web.xml`".
- **`pageScope`**, **`requestScope`**, **`sessionScope`** and **`applicationScope`**: A key-value map of variables in the page, request, session and application scopes respectively. They provide access to all the variables in the respective scope.

Operators

- Arithmetic Operators: Addition `'+'`, subtraction `'-'`, multiplication `'*'`, division `'/'` (or `div`), and modulus `'%'` (or `mod`).
- Comparison Operators: == (or eq), != (or ne), > (or `gt`), < (or `lt`), >= (or ge), <= (or `le`). The equality operator (== or eq), when applied to string, compare the contents of two strings (similar to Java's *`aString`*`.equals(`*`anotherString`*`)`).
- Boolean (Logical) Operators: && (or and), || (or or), ! (or not).
- Validation Operator `empty`: `empty` is a boolean unary operator checking for null value or empty string, e.g., `${empty param["user"]}` returns true if the request parameter "`user`" is `null` or an empty string. You could also use `${not empty ....}`.
- Shorthand if-else (`? :`): Similar to Java/C/C++, (*`test`* `?` *`trueExpression`* `:` *`falseExpression`*) returns the value of *`trueExpression`* if *`test`* resulted in true; or *`falseExpression`* otherwise.

Accessor Operators `'.'` and `[]`

The dot operator `'.'` can be used to access a property of an object (similar to `<jsp:getProperty>` but much easier to use). For example, `${aBean.aProperty}` returns `aProperty` of an object aBean. However, unlike Java's dot operator, it does not access the possibly private property (variable) directly, but indirectly via the getter `aBean.getAProperty()`. You can also access nested property, e.g., the expression `anObject.property1.property2` is equivalent to `anObject.getProperty1().getProperty2()`.

For simplicity, EL, unlike Java, does not throw exception. For example, `anObject.property1.property2` returns `null` if `anObject.property1` resulted in `null`, instead of throwing a `NullPointerException`.

The index operator `[]` can be used to access an element of an array; or a value of a collection (such as key-value map). For example, `${anArray[5]}`, `${aMap["aKey"]}`. Again, It returns `null` instead of throwing `ArrayIndexOutOfBound` exception if the index is beyond the array bound.

Although you can also use the dot operator as an indexing operator, e.g., `${aMap.aKey}` is the same as `${aMap["aKey"]}`, but that could be problematic in some situations and shall be avoided.

Precedence of Operators

The precedence of operators, form highest to lowest, is as follows:

- `[]` .

- [] .
- () (parentheses can be used to change the precedence)
- - (unary negate) not ! empty
- * / div % mod
- + - (binary subtract)
- < > <= >= lt gt le ge
- == != eq ne
- && and
- || or
- ? : (shorthand if-else)

Reserved Words

The followings are the reserved words in EL, which cannot be used as identifiers:

- and or not
- eq ne lt gt le ge
- true false null
- div mod empty instanceof

10.4 JSTL Core Actions

EL does not contain programming constructs such as if-else, loops, and setting variables. They are provided by the JSTL core actions, such as <c:set>, <c:out>, <c:if>, and <c:forEach>. For examples:

```
<c:set var="name" value="Peter"/>
  // Sets a variable called "name" in the default page scope, to literal value "Peter"
<c:set var="name" scope="session" value="${param['user']}" />
  // Sets a variable called "name" in session scope, to an EL expression
<c:out value="Hello, ${user.firstname} ${user.lastname}!" />
  // Prints the value, which consists of literals and EL expressions
```

To use the JSTL core library, we need to include this JSP taglib directive in the JSP page, similar to using any custom tag library. The tag library JAR-files "standard.jar" and "jstl.jar" shall be accessible by the web application.

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

The JSTL core library contains these actions:

- c:out
- c:set, c:remove
- c:if, c:choose, c:when, c:otherwise
- c:forEach, c:forTokens
- c:catch
- c:url, c:redirect, c:import, c:param

Output: <c:out>

Prints the result of a literal value or EL Expression.

```
// Prints the literal value or the result of the EL Expression
<c:out value="Literal|ELExpression" />
// Prints the default value, if the "value" is null.
<c:out default="defaultValue" value="Literal|ELExpression" />
// If the optional escapeXml is true, replace special HTML/XML characters
//  '<', '>', '&', ';' with their escape sequences such as &lt;, &gt;, &amp;, &quot;
<c:out value="Literal|ELExpression" escapeXml="true|false" />
```

For example,

```
// Prints a literal value
<c:out value="Hello, world" />
// Prints the result of an EL Expression
<c:out value="${user.firstname}" />
// Can concatenate literals and EL expressions
<c:out value="Hello, ${user.firstname} ${user.lastname}!" />
// Default for null value
<c:out default="everybody" value="${param['name']}" />
```

Setting/Removing Variable: <c:set>, <c:remove>

Sets the value of a variable in the chosen scope; or remove a variable. The attribute scope is optional, and default to page.

```
<c:set var="varName" scope="page|request|session|application" value="ELExpression" />
```

For examples,

```
<c:set var="name" value="Peter"/>
<c:set var="name" scope="session" value="${param['user']}" />
<c:set var="name" scope="session" value="${param.user}" />
<c:set var="area" value="${param['radius']*param['radius']*3.1416}" />
// Set a variable with a nesting a <c:out> with default
<c:set var="name"><c:out value="${param['user']}" default="everybody" /></c:set>
```

Conditional (or Selection): <c:if>, <c:choose>, <c:when>, <c:otherwise>

```
// Perform the body only if the expression returns true
<c:if test="expression"> ... </c:if>
```

```
// Assign the result of test to the variable
<c:if test="expression" var="varName" scope="scope"> ... </c:if>

// Similar to switch-case statement in Java/C/C++
<c:choose>
  <c:when test="ELExpression1"> ... </c:when>
  <c:when test="ELExpression2"> ... </c:when>
  ......
  <c:otherwise> ... </c:otherwise>
</c:choose>
```

Loops: <c:forEach>, <c:forTokens>

```
// Similar to for-each loop in JDK 1.5, to iterate thru a collection or array
<c:forEach var="varName" items="ELExpressionMapArray" >
  ......
</c:forEach>

// Similar to for-loop in Java
<c:forEach var="varName" begin="intExpression" end="intExpression" step="1|intExpression">
  ......
</c:forEach>

// forTokens is similar to Java's StringToken class for tokenizing string.
// The attribute delims specifies the delimiter.
<c:forTokens var="varName" items="aString" delims="delimiter">
  ......
</c:forTokens>
```

<c:catch>

Allows for rudimentary exception handling within a JSP page.

```
// Catch any exception
<c:catch> ...... </c:catch>

// Assign the exception to the variable
<c:catch var="varName"> ...... </c:catch>
```

<c:url>, <c:param>

Used to generate URLs, with optional request parameter specified in `<c:param>`. For example,

```
// Generate a URL
<a href="<c:url value='/checkout.jsp'/>">Check Out</a>

// Generate a URL with request parameter(s)
<c:url value="/search.jsp">
  <c:param name="searchWord" value="${searchWord}" />
</c:url>
```

The syntax is:

```
<c:url value="expression" context="expression" var="name" scope="scope">
  <c:param name="expression" value="expression" />
  ......
</c:url>
```

<c:import>

Similar to request-time `<jsp:include>` but more powerful. The optional "`var`" attribute causes the content to be stored in a variable (as string), rather than included in the current JSP page. The "`scope`" attribute controls the scope of this variable, and defaults to page scope.

<c:redirect>

Sends an HTTP redirect response to the client browser, equivalent to the `HttpServletResponse.sendRedirect()` method.

Example: [TODO]

10.5 JSTL function Actions

The JSTL function actions include:

- `fn:length`
- `fn:indexOf`, `fn:contains`, `fn:containsIgnoreCase`, `fn:endsWith`, `fn:startsWith`, `fn:substring`, `fn:substringAfter`, `fn:substringBefore`
- `fn:replace`, `fn:split`, `fn:join`
- `fn:escapeXml`
- `fn:toLowerCase`, `fn:toUpperCase`, `fn:trim` (??)

To use the JSTL function library, we need to include this JSP `taglib` directive in the JSP page:

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

Example

Unlike core actions that are used as tags, such as `<c:set>`, the function actions are used in EL expression such as `${fn:subtring(str, 3, 5}`.

```
<%@page contentType="text/html" pageEncoding="UTF-8" %>
<%@taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

```
      <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
      <title>Test JSTL functions actions</title>
   </head>
   <body>
      <c:set var="str" value="${empty param.str ? 'Hello, world!' : param.str}" />
      <c:if test="${fn:contains(str, 'Hello')}">
         <c:out value="'${str}' contains 'Hello'" /><br />
      </c:if>

      <c:out value="${fn:containsIgnoreCase(str, 'Hello') ? 'yes' : 'no'}" /><br />

      <c:out value="${fn:indexOf(str, 'a')}" /><br />

      ${fn:length(str)}<br />
      ${fn:substring(str, 3, 6)}<br />
      ${fn:substringAfter(str, ',')}<br />
      ${fn:substringBefore(str, ',')}<br />
      ${fn:endsWith(str, '!')}<br />
      ${fn:startsWith(str, 'H')}<br />

      <%-- Dot instead of ':' according to tld --%>
      <c:out value="${fn.toLowercase('TEST')}" /><br />
      <c:out value="${fn.toUppercase('test')}" /><br />
   </body>
</html>
```

[To Check: The function's TLD shows `fn.toLowercase()` - dot instead of `':'`.]

10.6 JSTL XML actions and XPath (XML Path Language)

The JSTL XML actions are meant for processing XML document. It uses the XPath (XML Path Language) for searching XML nodes, via pattern matching. Possible XML nodes include: root, element, attribute, comment, processing instruction, text, and namespace.

XPath (XML Path Language)

For example, suppose we have the following XML document "`ebookshop.xml`":

```
<?xml version="1.0" encoding="UTF-8"?>
<ebookshop>

<book category="Java">
  <title language="en">Java for Dummies</title>
  <author>Tan Ah Teck</author>
  <author>Mohammad Ali</author>
  <price>28.88</price>
</book>

<book category="Java">
  <title language="en">More Java for Dummies</title>
  <author>Tan Ah Teck</author>
  <price>38.88</price>
</book>

<book category="Game">
  <title language="cn">Mahjong 101</title>
  <author>Kumar</author>
  <author>Kevin Jones</author>
  <price>18.88</price>
</book>

</ebookshop>
```

We can use XPath to locate the nodes based on pattern matching. For example,

- `/ebookshop/book/title` returns all 3 titles.
- `/ebookshop/book/title[@language="en"]` returns the titles with attribute language="en"
- `/ebookshop/book[1]/author` returns the 2nd book's author.
- `/ebookshop/book[2]/author[1]` returns the 3rd book's 2nd author.
- `/ebookshop/book[price>20]/title` returns title with price more than 20.

JSTL XML actions

The JSTL xml actions include:

- `x:parse`, `x:transform`
- `x:param`
- `x:out`
- `x:set`
- `x:if`, `x:choose`, `x:when`, `x:otherwise`
- `x:forEach`

To use the JSTL XML library, we need to include this JSP `taglib` directive in the JSP page:

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

The XPath expression is used in the attribute `select="`*`XPathExpression`*`"`.

Example

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
```

```
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Test JSTL XML actions</title>
   </head>
   <body>
      <c:import url="ebookshop.xml" var="books" />
      <x:parse xml="${books}" var="dom" />

      <ul>
         <x:forEach var="book" select="$dom//book[price>20]" >
            <li>
               <x:out select="$book/title" />,
               <x:out select="$book/price" />,
               <x:forEach var="author" select="$book/author" >
                  <x:out select="$author" />,
               </x:forEach>
            </li>
         </x:forEach>
      </ul>

      <p><x:out select="$dom/ebookshop/book/title" /><p>
      <p><x:out select="$dom/ebookshop/book[2]/title" /><p>

   </body>
</html>
```

The `<c:import>` reads the entire XML document into variable `books`. The `<x:parse>` is then used to parse the XML document into a DOM tree. We can use use XPath expression in the `select` attribute to select nodes. We can use `<x:forEach>` to iterate thru the nodes, and `<x:out>` to output.

Notes: If you encounter error "`java.lang.ClassNotFoundException: org.apache.xpath.VariableStack`", download the xalan from http://mirror.nus.edu.sg/apache/xml/xalan-j/, and copy the `xalan.jar` into Tomcat's `lib` (or the application's `WEB-INF\lib`).

[TODO] more example on xlst.

10.7 JSTL i18n fmt Actions

The JSTL internationalization (i18n) library provides actions to produce customized web pages based on the client's *locale* and *time zone*. A locale includes language, formatting of dates, numbers, and currency amounts, etc.

The HTTP request header includes an `Accept-Language`, which is made available to servlet via `getLocale()` and `getLocales()` methods of the `ServletRequest` class. However, there is no timezone information provided by the HTTP request header.

The JSTL i18n actions include:

- `fmt:setLocale`
- `fmt:setBundle`, `fmt:bundle`, `fmt:message`, `fmt:param`
- `fmt:timeZone`, `fmt:setTimeZone`, `fmt:parseDate`, `fmt:parseNumber`, `fmt:formatDate`, `fmt:formatNumber`
- `fmt:requestEncoding`

To use the JSTL i18n library, we need to include this JSP `taglib` directive in the JSP page:

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

<fmt:setLocale>

The locale used by the JSTL is determined by the `Accept-Language` request header. If no such header presents, it should the default value of your system. You could override the locale setting via `<fmt:setLocale>`, as follows:

```
<fmt:setLocale value="expression" [scope="page|request|session|application"] [variant="expression"] />
```

The attribute "`value`" should be either a locale name string, or an instance of the `java.util.Locale` class. A locale name is made up of a two-letter lowercase ISO country code, and a two-letter uppercase ISO language code, e.g., `en_US`, `fr_CA`, etc. The optional attribute "`variant`" can be used to specify the browser or OS, e.g., `MAC`, `WIN`.

<fmt:setBundle>, <fmt:bundle>, <fmt:message>, <fmt:param>

The `<fmt:setBundle>` action sets a default resource bundle for use by `<fmt:message>` tags within a particular scope, by specifying the *basename* of resource bundle as follows. The value for the `basename` attribute should not include any localization suffixes or filename extensions.

```
<fmt:setBundle basename="expression" var="varName" [scope="scope"] />
```

The `<fmt:bundle>` action is similar to `<fmt:setBundle>`, but it specifies the resource bundle for use by the `<fmt:message>` nested within its body content.

`<fmt:message>` allows you to retrieve text messages from a locale-specific resource bundle and display it on a JSP page. Furthermore, you could include parameterized values in text messages to customize the content dynamically via `<fmt:param>` tag.

```
<fmt:message key="expression" bundle="expression"
      var="name" scope="scope"/>

<fmt:message key="expression" bundle="expression"
      var="name" scope="scope">
   <fmt:param value="expression"/>
   ......
</fmt:message>
```

Example

Resource Bundle Properties: Create the following resource bundle (properties file) "`LocalString_en.properties`" and "`LocalString_fr.properties`". Save under `<contextRoot>\WEB-INF\classes\mypkg`.

(NetBeans) "Files" panel ⇒ classes ⇒ right-click mypkg ⇒ New ⇒ Other ⇒ Properties File.

`<contextRoot>\WEB-INF\classes\mypkg\LocalString_en.properties`

```
title=Hello World!
greeting=Hello {0} {1}!
```

<contextRoot>\WEB-INF\classes\mypkg\LocalString_fr.properties

```
title=Salut le Monde!
greeting=Bonjour {0} {1}!
```

You may also create a *default* "`LocalString.properties`".

`TestLocale.jsp`: The following JSP page takes 3 optional request parameters: lang=xx, firstname=xx and lastname=xx, and displays the locale strings.

```
<%@page contentType="text/html" pageEncoding="UTF-8" %>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Test Locale String in Resource Bundle</title>
   </head>
   <body>
      <c:set var="lang" value="${empty param.lang ? 'en' : param.lang}" />

      <fmt:setLocale value="${lang}" />
      <fmt:setBundle basename="mypkg.LocalStrings" var="bundle" />

      <h1><fmt:message key="title" bundle="${bundle}" /></h1>

      <c:set var="firstname" value="${empty param.firstname ? 'Peter' : param.firstname}" />
      <c:set var="lastname" value="${empty param.lastname ? 'Paul' : param.lastname}" />
      <fmt:bundle basename="mypkg.LocalStrings">
         <p><fmt:message key="greeting" >
               <fmt:param value="${firstname}" />
               <fmt:param value="${lastname}" />
            </fmt:message></p>
      </fmt:bundle>
   </body>
</html>
```

The first `<fmt:message>` uses the resource bundle defined via `<fmt:setBundle>` tag. The second `<fmt:message>` is nested within a `<fmt:bundle>` tag, and uses parametric value set via `<fmt:param>`.

You will see ???*keyname*??? if the system cannot locate your properties file or message matching the key.

`<fmt:timeZone>, <fmt:setTimeZone>, <fmt:parseDate>, <fmt:parseNumber>, <fmt:formatDate>, <fmt:formatNumber>`

[TODO]

10.8 JSTL `sql` Actions

The JSTL sql actions include:

- Create Database Connection: `<sql:setDataSource>`
- Query and Update: `<sql:query>`, `<sql:update>`, `<sql:param>`, `<sql:dateParam>`
- Transaction Management: `<sql:transaction>`

To use the JSTL function library, we need to include this JSP `taglib` directive in the JSP page:

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

Example:

Let us begin JSTL sql actions with an example. I shall assume that you are familiar with SQL and JDBC.

`Setting Up Database`: Create a database called "ebookshop", with a table called "`books`" with 5 columns: `id`, `title`, `author`, `price`, `qty`. You could run the following SQL statements on a MySQL system:

```
create database if not exists ebookshop;

use ebookshop;

drop table if exists books;
create table books (
   id int,
   title varchar(50),
   author varchar(50),
   price float,
   qty int,
   primary key (id));

insert into books values (1001, 'Java for dummies', 'Tan Ah Teck', 11.11, 11);
insert into books values (1002, 'More Java for dummies', 'Tan Ah Teck', 22.22, 22);
insert into books values (1003, 'More Java for more dummies', 'Mohammad Ali', 33.33, 33);
insert into books values (1004, 'A Cup of Java', 'Kumar', 55.55, 55);
insert into books values (1005, 'A Teaspoon of Java', 'Kevin Jones', 66.66, 66);

select * from books;
```

`TestDBQuery.jsp`: Write the following JSP page (called "`testDBQuery.jsp`") to query the database. Assume that the MySQL is running on port 3306, with a user called "`myuser`" with password.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Test JSTL sql actions</title>
```

```
</head>
<body>
  <sql:setDataSource var="ds"
    url="jdbc:mysql://localhost:3306/ebookshop" user="myuser" password="xxxx"/>

  <sql:query sql="select * from books" var="rset" dataSource="${ds}" />

  <table border="1">
    <tr>
      <th>ID</th>
      <th>Title</th>
      <th>Author</th>
      <th>Price</th>
      <th>Qty</th>
    </tr>
    <c:forEach var="row" items="${rset.rows}">
      <tr>
        <td><c:out value="${row.id}" /></td>
        <td><c:out value="${row.title}" /></td>
        <td><c:out value="${row.author}" /></td>
        <td><c:out value="${row.price}" /></td>
        <td><c:out value="${row.qty}" /></td>
      </tr>
    </c:forEach>
  </table>
</body>
</html>
```

The above JSP page uses `<sql:setDataSource>` to create a database connection called "ds" via a database-URL `jdbc:mysql://host:port/defaultDatabase`. It then use `<sql:query>` to issue a "`SELECT * FROM books`", and keep the query result set in variable `rset`. The core action `<c:forEach>` is used to iterate thru all the rows; `<c:out>`'s are used to print the cells in a HTML table.

`<sql:setDataSource>`

You can use `<sql:setDataSource>` to create a database connection via a database-URL, as in the above example. The syntax is:

```
// Create a database connection via a database-url, and
//   assign to the specified scoped variable.
<sql:setDataSource url="databaseURL" var="varName"
    [scope="page|request|session|application"]
    [user="expression"] [password="expression"] [driver="expression"] />
```

JSBC's `Datasource` is a factory for obtaining database connections. `DataSource` supports so-called *connection pooling* to reduce the overhead associated with the creating and initializing connections. The `<sql:setDataSource>` syntax for using `DataSource` connection pooling is as follow:

```
// Get a connection from the DataSource connection pool via JNDI lookup,
//   and assign to a scoped variable.
<sql:setDataSource dataSource="JDNIName" var="varName"
    [scope="page|request|session|application"] />
```

Database Connection Pooling (DBCP)

Let's try out the connection pooling. Tomcat provides built-in support for datasource with connection pooling, which are made available to applications through the Java Naming and Directory Interface (JNDI).

`context.xml`: Create the following configuration file "`context.xml`", and save under "*<contextRoot>*`\META-INF`". [For server-wide configuration, put the `<Resource>` element under `<GlobalNamingResources>` in `<CATALINA_HOME>\conf\server.xml`.]

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
  <!--
    maxActive: Maximum number of dB connections in pool. Set to -1 for no limit.
    maxIdle: Maximum number of idle dB connections to retain in pool. Set to -1 for no limit.
    maxWait: Maximum milliseconds to wait for a dB connection to become available
             Set to -1 to wait indefinitely.
  -->
  <Resource name="jdbc/TestDB" auth="Container" type="javax.sql.DataSource"
     maxActive="100" maxIdle="30" maxWait="10000" removeAbandoned="true"
     username="myuser" password="xxxx" driverClassName="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost:3306/ebookshop" />
</Context>
```

`web.xml`: Next, modify the application's deployment descriptors in "*<contextRoot>*`\WEB-INF\web.xml`", with a datasource references the JNDI resource name "`jdbc/TestDB`" defined earlier.

```
<web-app ......>
   .......
   <resource-ref>
      <description>DB Connection Pool</description>
      <res-ref-name>jdbc/TestDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
      <res-sharing-scope>Shareable</res-sharing-scope>
   </resource-ref>
</web-app>
```

`TestDBQueryDBCP.jsp`: The following JSP page gets a connection from the pool, and use the connection to issue a `SELECT` query.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Test JSTL sql actions with Connection Pooling</title>
</head>
```

```
</head>
<body>
  <sql:setDataSource var="ds" dataSource="jdbc/TestDB" />
  <sql:query sql="select * from books" var="rset" dataSource="${ds}" />

  <table border="1">
    <tr>
      <th>ID</th>
      <th>Title</th>
      <th>Author</th>
      <th>Price</th>
      <th>Qty</th>
    </tr>
    <c:forEach var="row" items="${rset.rows}">
      <tr>
        <td><c:out value="${row.id}" /></td>
        <td><c:out value="${row.title}" /></td>
        <td><c:out value="${row.author}" /></td>
        <td><c:out value="${row.price}" /></td>
        <td><c:out value="${row.qty}" /></td>
      </tr>
    </c:forEach>
  </table>
</body>
</html>
```

<sql:query>

Once the database connection is established, we can use <sql:query> (similar to JDBC's executeQuery()) and <sql:update> (executeUpdate()).

<sql:query> issues a SELECT query via the "datasource", and keeps the returned result set in the scoped variable. The syntax is:

```
<sql:query sql="SqlExpression" dataSource="dsExpression"
      var="name" [scope="page|request|session|application"]
      [maxRows="intExpression"] [startRow="intExpression"] />
```

The returned result set implements javax.servlet.jstl.sql.Result interface, with the following properties:

- **rows**: An array of SortedMap of column name and value.
- **rowsByIndex**: An array of array, each item corresponding to a single row in the result set.
- **columnNames**: An array of strings of column names.
- **rowCount**: Total number of rows in the result set.
- **limitedByMaxRows**: A boolean, true if result set is limited by maxRows attribute.

You can also use the parameterized PreparedStatement, with parameters specified in <sql:param> and <sql:dateParam> tags.

```
// Use parameterized PreparedStatement
<sql:query sql="sqlExpression" dataSource="dsExpression"
      var="varName" [scope="scope"]
      [maxRows="intExpression"] [startRow="intExpression"] >
   <sql:param value="expression" />
   ......
</sql:query>

// Place the SQL statement in the body
<sql:query dataSource="expression"
      var="varName" [scope="scope"]
      [maxRows="intExpression"] [startRow="intExpression"] >
   SQLStatement
   <sql:param value="expression" />
   ......
</sql:query>
```

For example, the following SQL statement contains a parameter (denoted as '?'), which will be replaced by the parameter in <sql:param>.

```
<sql:setDataSource var="ds" dataSource="jdbc/TestDB" />
<sql:query sql="select * from books limit ?" var="rset" dataSource="${ds}" >
   <sql:param value="${2}" />
</sql:query>
```

The syntax for <sql:param> and <sql:dateParam> (for date type) is as follows:

```
<sq:param value="expression" />
<sql:dateParam value="expression" type="date|time|timestamp" />
```

<sql:update>

<sql:update> issue a JDBC's executeUpdate(), which returns an an instance of java.lang.Integer (instead of a result set). The syntax is similar to <sql:query>:

```
<sql:update sql="SqlExpression" dataSource="dsExpression"
   var="varName" [scope="scope"] />

<sql:update sql="sqlExpression" dataSource="dsExpression"
      var="name" [scope="scope"] >
   <sql:param value="expression" />
   ......
</sql:update>

<sql:update dataSource="dsExpression"
      var="varName" [scope="scope"] >
   SQLStatement
   <sql:param value="expression" />
   ......
</sql:update>
```

Example:

```
<sql:setDataSource var="ds" dataSource="jdbc/TestDB" />
<sql:update sql="update books set qty = 0" var="rcode" dataSource="${ds}" />
<p><c:out value="${rcode} rows affected" /></p>
```

Managing Transaction <sql:transaction>

A *transaction* is a sequence of database operations that must either succeed or fail as a group. In JSTL, you can wrap a series of queries and updates into a transaction by nesting the corresponding <sql:query> and <sql:update> actions in the body content of a <sql:transaction> tag. The syntax for <sql:transaction> is as follows:

```
<sql:transaction dataSource="dsExpression" [isolation="isolationLevel"] >
   <sql:query ... /> | <sql:update ... />
   ......
</sql:transaction>
```

The attribute isolation specifies the *isolation level* for the transaction, and may be either read_committed, read_uncommitted, repeatable_read, or serializable.

Example [TODO]

11. JSP in XML Syntax

JSP pages with scripting elements (<% ... %>) are not well-formed XML documents. JSP 2.0 (??) supports JSP in XML-compliant syntax, which is convenient for applications involving AJAX and Web Services.

The classical JSP scripting tags in the form of <% ... %> are not XML-compliant. A second set of XML-compliant tag with a prefix of "jsp" is defined as follows:

- JSP Scriptlet: <jsp:scriptlet>
- JSP Expression: <jsp:expression>
- JSP Declaration: <jsp:declaration>
- JSP Directive: <jsp:directive>, <jsp:directive.page>, <jsp.directive.include>, <jsp.directive.import>
- <jsp:output>
- <jsp:root>, <jsp:element>, <jsp:attribute>, <jsp:body>

JSP pages in XML syntax is called JSP document, with a file extension of ".jspx" (JSP XML document).

Example: Hello-world

The following JSP document "hello.jspx" prints "Hello world!".

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">

   <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>

   <jsp:element name="text">
      <jsp:attribute name="lang">EN</jsp:attribute>
      <jsp:body>Hello World!</jsp:body>
   </jsp:element>

</jsp:root>
```

Notes:

- Line 1 <?xml ... ?> identifies this file as a XML document.
- A JSP XML document begins with a root element called <jsp:root>.
- The <%@page ... %> directive is replaced by <jsp:directive.page> element.
- You can use <jsp:element> with nested elements <jsp:attribute> and <jsp:body>.

Example

We shall rewrite the "first.jsp" into XML syntax as "first.jspx" as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>

  <html>
  <head><title>First JSP Document</title></head>

  <body>
    <jsp:scriptlet>
      double num = Math.random();
      if (num > 0.95) {
    </jsp:scriptlet>
        <h2>You will have a luck day!</h2><p>(<jsp:expression>num</jsp:expression>)</p>
    <jsp:scriptlet>
      } else {
    </jsp:scriptlet>
        <h2>Well, life goes on ... </h2><p>(<jsp:expression>num</jsp:expression>)</p>
    <jsp:scriptlet>
      }
    </jsp:scriptlet>
      <jsp:text><![CDATA[<a href="]]></jsp:text>
      <jsp:expression>request.getRequestURI()</jsp:expression>
      <jsp:text><![CDATA["><h3>Try Again</h3></a>]]></jsp:text>
  </body>
  </html>
</jsp:root>
```

Notes:

- If the texts contains special XML characters (such as <, >, &, "), you need to enclose within <![CDATA[*texts*]]> (Character Data Section). The CDATA section will be sent to the client, uninterpreted (which defeats the purpose of validation).
- To support the full validation of the resultant HTML document, you could include the proper DOCTYPE in the generated HTML document by replacing the <html> tag with a <jsp:output>

- To support the full validation of the resultant HTML document, you could include the proper `!DOCTYPE` in the generated HTML document by replacing the `<html>` tag with a `<jsp:output>` to print the `!DOCTYPE` and modify the `<html>` tag as follows:

```
<jsp:output
  doctype-root-element="html"
  doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
  doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" />
<html xmlns="http://www.w3.org/1999/xhtml">
```

- To produce newline in the generated HTML file, you need to use `<jsp:text>...</jsp:text>`, with newline in the body.
- Take note that JSP with scripting element is hard to write, and equally write to read. You should use JSTL/EL or custom tags.

Taglib

To use JSTL taglib, instead of using `<%@taglib ... %>` JSP directive, you declare XML namespaces in `<jsp:root>` element `xmlns` attributes as follows:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:c="http://java.sun.com/jsp/jstl/core"
          xmlns:sql="http://java.sun.com/jsp/jstl/sql"
          xmlns:x="http://java.sun.com/jsp/jstl/xml"
          xmlns:fmt="http://java.sun.com/jsp/jstl/fmt"
          xmlns:fn="http://java.sun.com/jsp/jstl/functions"
          version="2.1">
```

12. JSP Fragment/Segment

[TODO]

REFERENCES & RESOURCES

1. JSP (JavaServer Pages) Home Page @ http://www.oracle.com/technetwork/java/javaee/jsp/index.html, and JSP Developer Site @ http://jsp.java.net.
2. JSP 1.1, 1.2, 2.0, 2.1, 2.2 Specifications. JSTL 1.0, 1.2 Specifications. EL 2.1, 2.2 Specifications.
3. JSTL (JSP Standard Tag Libraries) 1.2 developers @ http://jstl.java.net, and JSTL 1.1 Apache Jakarta's Taglibs @ http://jakarta.apache.org/taglibs/index.html.
4. EL Developers @ http://uel.java.net.
5. Apache Tomcat Server @ http://tomcat.apache.org.
6. Java Servlets Home Page @ http://java.sun.com/products/servlet. Servlet Developers @ http://java.net/projects/servlet/.
7. Glassfish developers @ http://glassfish.java.net.
8. RFC2616 "Hypertext Transfer Protocol HTTP 1.1", W3C, June 1999.
9. HTML 4.01 Specification, W3C Recommendation, 24 Dec 1999 @ http://www.w3.org/TR/html401, and HTML 5 Draft Specification @ http://www.w3.org/TR/html5.
10. Marty Hall, "Core Servlets and JavaServer Pages", vol.1 (2nd eds, 2003) and vol. 2 (2nd eds, 2006), Prentice Hall.
11. Andy Grant, "JSP 2.0 Simple Tags Explained" @ http://www.sitepoint.com/jsp-2-simple-tags/.
12. Mark Kolb, "A JSTL primer, Part 1 to Part 4" @ http://www.ibm.com/developerworks/java/library/j-jstl0211/index.html.
13. The Java EE 6 Tutorial Volume 1, December 2009 @ http://java.sun.com/javaee/6/docs/tutorial/doc/.
14. The Java EE 5 Tutorial, Chapter 5 JavaServer Pages Technology, October 2008 @ http://download.oracle.com/javaee/5/tutorial/doc/bnagx.html.
15. The J2EE 1.4 Tutorial, Chapter 12 JavaServer Pages Technology, December, 2005 @ http://java.sun.com/j2ee/1.4/docs/tutorial/doc/.
16. The J2EE 1.3 Tutorial "JavaServer Pages Technology" @ http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPIntro.html.
17. Java EE 6 Technologies @ http://www.oracle.com/technetwork/java/javaee/tech/index-jsp-142185.html.
18. Java EE 5 Technologies @ http://www.oracle.com/technetwork/java/javaee/tech/javaee5-jsp-135162.html.
19. java.net - The Source for Java Technology Collaboration @ http://www.java.net.