Definitions and some examples

JSP

Java **S**erver **P**ages

By Hamid Mosavi-Porasl

Java Server Pages (JSP) is a technology based on the Java language and enables the development to dynamic web sites. JSP was developed by Sun Microsystems to allow server side development. JSP files are HTML files with special tags containing Java source code that provide the dynamic content.

1 JSP compared to servlet

A Servlet is a Java class that provides special server side services. It is hard work to write HTML code in Servlets. In Servlets you need to have lots of println statements to generate HTML.

2 Steps required for a JSP request

1. The user goes to a web site made using JSP. The web browser makes the request via Internet.

2. The JSP request gets sent to the web Server

3. The Web server recognizes that the file required is special (.jsp), therefore passes the JSP file to the JSP Servlet Engine.

4. If the JSP file has been called the first time, the JSP file is parsed, otherwise go to step 7.

5. The next step is to generate a special Servlet from the JSP file. All the HTML required is converted to println statements.

6. The Servlet source code is compiled into a class.

7. The Servlet is instantiated, calling the Init and service methods.

8. HTML from the Servlet output is sent via the Internet

9. HTML results are displayed on the user's web browser.

3 Simple JSP Page

```
<html>
<head>
<title>A simple JSP page</title>
</head>
<body>
<%@ page language="java" %>
<% System.out.println("A simple JSP page"); %>
</body>
</html>
```

Type the code above into a text file. Name that to simpleJspPage.jsp. Paste that to correct directory.

4 There are five main tags in JSP

1. Declaration tag
2. Expression tag
3. Directive tag
4. Scriptlet tag
5. Action tag

4.1 Declaration tag (<%! %>)

This tag allows the developer to declare variables or methods.
For example:

```
<%!
    private int counter=0;
    private String get Account(int accountNo);
%>
```

4.2 Expression tag (<%= %>)

This tag allows the developer to embed any Java expression and is short for out.println()
A semicolon (;) doesn't appear at the end of the code inside the tag.
For example:
Date: ***<%= new java.util.Date() %>***

4.3 Directive tag (<%@ directive … %>)

A JSP directive tag gives special information about the page to the JSP engine.
There are three main types of directives:

1) Page – processing information for this page
2) Include – files to be included
3) Tag library – tag library to be used in this page

Directives don't produce any visible output when the page is requested but change the way the JSP engine processes the page.

4.3.1 Page directive:

This directive has 11 optional attributes that provides the JSP Engine with special processing information:

language	Language the file uses	<%@page language="java"%>
Extends	Superclass used by the JSP engine for the translated Servlet	<%@page extends="com.taglib…"%>
import	Import all the classes in a java package into the current JSP page. This allows the JSP page to use other java classes.	<%@page import="java.util"%>

	Following packages are implicitly imported. java.lang.*; javax.servlet.*; javax.servlet.*; javax.servlet.jsp.*; javax.servlet.http.*;	
Session	Does the page make use of sessions. By default all JSP pages have session data available. There are performance benefits to switching session to false.	Default is set to true
Buffer	Controls the use of buffered output for a JSP page. Default is 8 kb	<%@page buffer=”none”%>
autoFlush	Flush output buffer when full	<%@page autoFlush=”true”%>
IsThreadSafe	Can the generated Servlet deal with multiple requests? If true a new thread is started so requests are handled simultaneously	
Info	Developer uses info attribute to add information/document for a page. Typically used to add author, version, copyright and date info	<%@page info=”visualbuilder.com test page, copyright 2001.”%>
ErrorPage	Different page to deal with errors. Must be URL to error page.	<%@page errorPage=”/error/error.jsp”%>
IsErrorPage	This flag is set to true to make a JSP page a special Error page. This page has access to the implicit object exception	
ContentType	Set the mime type and character set of the JSP	

Table 1 - 11 optional attributes for JSP directive

4.3.2 Include directive

Allows a JSP developer to include contents of a file inside another. Typically include files are used for navigation, tables, headers and footers that are common to multiple pages.

Two examples of using include files:

This includes the html from privacy.html found in the include directory into the current JSP page.

<%@ include file="include/privacy.html %>
or to include a navigation menu (JSP file) found in the current directory.
<%@ include file="navigation.jsp%>

4.3.3 Tag Lib directive

A tag lib as a collection of custom tags that can be used by the page.
<%@ taglib uri="tag library URI" prefix="tag Prefix" %>
Custom tags were introduced in JSP1.1 and allow JSP developers to hide complex server side code from web designers.

4.4 Scriptlet tag (<% ….%>)

Between <% and %> tags, any valid java code is called a Scriptlet. This code can access any variable or bean declared.

```
<%
        String username="visualbuilder";
        Out.println(username);
%>
```

4.5 Action tag

There are three main roles of action tags:

1) enable the use of server side Java beans
2) transfer control between pages
3) browser independent support for applets

5 Java Beans

A java bean is a special type of class that has a number of methods. The JSP page can call these methods so can leave most of the code in these java beans. For example, if you wanted to make a feedback from the automatically sent out an email. By having a JSP page with a form, when the visitor presses the submit button this sends the details to a java bean that sends out the email. This way there would be no code in the JSP page dealing with sending emails (Java Mail API) and your Java bean could be used in another page.

To use a java bean in a JSP page we use the following syntax:
<jsp: usebean id="…" scope="application" class="com…"/>

The following is a list of Java bean scopes:

page – valid until page completes

request – bean instance lasts for the client request
session – bean lasts for the client session
application – been instance created and lasts until application ends.

6 Dynamic JSP Include

We have seen how a file can be included into a JSP using an include Directive:
<%@ include file=”include/privacy.html%>
This is useful for including common pages that are shared and is included at compile time. To include a page at run time we should use dynamic JSP includes.
<jsp:include page=”URL” flush=”true”/>

6.1 An example:

In following example above mentioned JSP tags are used:

```
<html>
<head>
<title> JSP TAGS</title>
</head>
<body> JSP TAGS example
<BR>
<%!
    String sitename=”www.porasl.com”;
    int counter=0;

 private void Counter()
{
    counter++;
}
%>

website of the day is
<%=sitename%>
<BR>
Page accessed <%=counter%>
</body>
</html>
```

7 Action and Implicit Objects

Application specific classes can be initiated and values from method calls can be inserted into JSP output. Network resources and repositories such as databases can be accessed to store and retrieve data for use by JSP pages. In addition to objects such as these, which are completely under the control of the developer, the JSP container also exposes a number of its internal objects to the page author. These are referred to *implicit objects*, because their availability in a JSP page is automatic. The developer can assume that

these objects are present and accessible via JSP scripting elements. Each implicit object must adhere to a corresponding API, in the form of a specific Java class or interface definition. Thus it will either be an instance of the class or interface, or of an implementation-specific subclass.

Object	Class or interface	Description
page	javax.servlet.jsp.HttpJspPage	Page's servlet instance
config	javax.servlet.ServletConfig	Servlet configuration data
request	javax.servlet.http.HttpServletRequest	Request data, including parameters
response	javax.servlet.http.HttpServletResponse	Response data
out	javax.servlet.jsp.JspWriter	Output stream for page content
session	javax.servlet.http.HttpSession	User-specific session data
application	javax.servlet.ServerContetext	Data shared by all application pages
pagecontent	javax.servlet.jsp.PageContext	Context data for page execution
exception	java.lang.throwable	Uncaught error exception

Table 2 Nine implicit objects provided by JSP

Beyond this functional categorization, four of the JSP implicit objects – *request, session application* and *pageContext* – have something else in common: the ability to store and retrieve arbitrary attribute values. By setting and getting attribute values, these objects are able to transfer information between and among JSP pages and servlets as a simple data-sharing mechanism.

The standard methods for attribute management provided by the classes and interfaces of these four objects are summarized in following table. Note that attribute keys take the form of Java String objects, while their values are referenced as instances of *java.lang.Object.*

Method	Description
setAttribute(key,value)	Associates an attribute value with a key (i.e , a name).
getAttributeNames()	Retrievers the names of all attributes associated with the session
getAttribute(key)	Retrieves the attribute value associated with the key
removeAttribute(key)	Removes the attribute value associated with the key

Table 3 Common methods for storing and retrieving attribute values

7.1 Servlet-related objects

The two JSP implicit objects in this category are based on the JSP page's implementation as aservlet. The page implicit object represents the servlet itself while the config object stores the servlet's initialization parameters.

7.1.1 Page object

The page object represents the JSP page itself or more specifically, an instance of the servlet class into which the page has been translated. It may be used to call any of the methods defined by that servlet class.

Here is an example page fragment that utilizes this implicit object:

```
<%@ page info=”page implicit demonstration.”%>
page info:
<%=((javax.servlet.jsp.HttpJspPage) page).getServletInfo()%>
```

This expression will insert the value of the page’s documentation string into the output from the page. In this example, note that because the servlet class varies from one page to another, the standard type for the page implicit object is the default Java type for nonprimitive values, java.lang.object. In order to access methods defined by the javax.servlet.jsp.HttpJspPage interface, the page object must first be cast to that interface.

After running following HTML code:

```
<html><head>
</head>
<body>
<%@ page info="page implicit demonstration."%>
page info:
<%=((javax.servlet.jsp.HttpJspPage) page).getServletInfo()%>
</body>
</html>
```

We get following output after running (page.jsp):

page info: page implicit demonstration.

7.1.2 Config object

The config object stores servlet configuration data – in the form of initialization parameters – for the servlet into which a JSP page is compiled. Because JSP pages are seldom written to interact with initialization parameters, this implicit object is rarely used in practice. This object is an instance of the javax.servlet.ServletConfig interface.

Method	Description
getInitParameterNames()	Retrieves the names of all initialization parameters
getInitParameter(name)	Retrieves the value of the named initialization parameter

Table 4 Methods of javax.servlet.ServletConfig Interface for accessing Initialization parameters

Due to its role in servlet initialization, the config object tends to be most relevant in a page’s initialization routine. Consider the following variation on the sample jspInt() method:

```
<%! Static private DbConnectionPool pool=null;
```

```
        public void jspInit(){
                if (pool == null){
                        String username = config.getInitParameter("username");
                        String password = config.getInitParameter("password");
    }
} %>
```

In this case rather than storing the username and password values directory in the JSP page, they have been provided as initialization parameters and are accessed via the config object.

Values for initialization parameters are specified via the deployment descriptor file of a web application.

7.1.3 Input/output

These implicit objects are focused on the input and output of a JSP page. More specifically, the request object represents the data coming into the page while the response object represents its result. The out implicit object represents the actual output stream associated with the response, to which the page's content is written.

7.1.3.1 Request object

The request object represents the request that request that triggered the processing of the current page. For HTTP requests, this object provides access to all of the information associated with a request including its source the requested URL and any headers, cookies, or parameters associated with the request. The *request* object is required to implement the *javax.servlet.ServletRequest* interface. When the protocol is HTTP, as is typically the case, it must implement a subclass of this interface, *javax.servlet.http.HttpServletRequest*.

The request object is one of the four JSP implicit objects the support attributes, by means of the methods presented in Table 4. The HttpServletRequest interface also includes methods for retrieving request parameters and HTTP headers.

Among the most common uses for the request object are looking up parameter values and cookies. Here is a page fragment illustrating the use of the *request* object to access a parameter value:

```
<% String xStr=request.getParameter("number");%>
try{ long x =Long.parseLong(xStr); %>
Factorial result: <% =x %>! =<%=fact(x)%>
<% } catch(NumberFormatException e){ %>
Sorry, the <b> number</b> parameter does not specify an integer value.
<% } %>
```

In this example the value of the *number* is fetched from the request. Note that all parameter values are stored as strings so conversion is required before it may be used as a number. If the conversion succeeds, this value is used to demonstrate the factorial function. If not, an error message is displayed.

When utilizing the <jsp:forward> and <jsp:include> action described at the end of this chapter, the request object is also often used for storing and retrieving attributes in order to transfer data between pages.

Method	Description
getParameterNames()	Returns the names of all request parameters
getParameter(name)	Returns the first (or primary) value of a single request parameter
getParaneterValues(name)	Retrieves all of the values for a single request parameter

Table 5 - Methods of the javax.servlet.http.HttpServletRequest interface for accessing request

Method	Description
getHeaderNames()	Retrieves the names of all of headers associated with the request
getHeader(name)	Returns the value of a single request header, as a string
getHeaders(name)	Returns all of the values for a single request header
getIntHeader(name)	Returns the value of a single request header, as an integer
getDateHeader(name)	Returns the value of a single request header, as a date
getCookies()	Retrieves all of the cookies associated with the request

Table 6 - Methods of the javax.servlet.http.HttpServletRequest interface for retrieving request headers

Method	Description
getMethod()	Returns the HTTP (e.g., GET, POST) method for the request.
getRequestURL()	Returns the request URL up to but not including any query string
getQueryString()	Returns the query string that follows the request URL, if any
getSession(flag)	Retrieves the session data for the request (i.e., the session implicit object) optionally creating it if it doesn't already exist
getRequestDispatcher(path)	Creates a request dispatcher for the indicated local URL
getRemoteHost()	Returns the fully qualified name of the host that sent the request
getRemoteAddr()	Returns the network address of the host that sent the request
getRemoteUser()	Return the name of user that sent the request, if known.
getSession(flag)	Retrieves the session data for the request (i.e., the session implicit object), optionally creating it if it doesn't already exist.

Table 7 – Miscellaneous methods of the javax.servlet.http.HttpServletRequest interface

7.1.3.2 Response object

The response object represents the response that will be sent back to the user as a result of processing the JSP page. This object implements the *javax.servlet.ServletResponse* interface. If it represents an HTTP response, it will furthermore implement a subclass of this interface, the javax.servlet.http.HttpServletResponse interface.

Method	Description
getContentType()	Set the MIME type and, optionally, the character encoding of the response's contents.
getCharacterEncoding()	Returns the character encoding style set for the response's contents

Table 8 – Methods of the javax.servlet.http.HttpServletResponse interface for specifying content

Method	Description
addCookie(cookie)	Adds the specified cookie to the response
containsHeader(name)	Checks whether the response includes the named header
setHeader(name, value)	Assigns the specified string value to the named header
setIntHeader(name, value)	Assigns the specified integer value to the named header
setDateHeader(name value)	Assigns the specified date value to the named header
addHeader(name value)	Adds the specified string value as a value for the named header
addIntHeader(name, value)	Adds the specified integer value as a value for the named header.
addDateHeader(name, date)	Adds the specified date value as a value for the named header.

Table 9 Methods of the javax.servlet.http.HttpServletResponse Interface for setting response headers

Method	Description
SetStatus(code)	Sets the status code for the response (for non-error circumstances).
SetError(status,msg)	Sets the status code and error message for the response.
SendRedirect(url)	Sends a response to the browser indicating it should request an alternate (absplute) URL.

Table 10 – Response code methods of the javax.servlet.http.HttpServletResponse Interface

Method	Description
EncodeRedirectURL(url)	Encodes a URL for use with the sendRedirect() method to include session information.
EncodeURL(name)	Encodes a URL used in a link to include session information.

Table 11– Methods of the javax.servlet.http.HttpServletResponse interface for performing URL rewriting

Below is a scriplet that uses the *response* object to set various headers for preventing the page from being cached by a browser:

```
<% response.setDateHeader("Expires",0);
    response.setHeader("Pragma" "no-cache");
   if(request.getProtocol().equals("HTTP/1.1")){
          response.setHeader("Cache-Control", "no-cache");
          }
%>
```

The scriptlet first sets the Expires header to a date in the past. This indicates to the recipient that the page's contents have already expired, as a hint that its contents should not be cached.

7.1.3.3 Out object

This implicit object represents the output stream for the page, the contents of which will be sent to the browseras the body of its response. The out object is an instance of the *javax.servlet.jsp.JspWriter* class. This is an abstract class that extends the standard *java.io.Writer* class. This an abstract class that extends the standard *java.io.writer* class. In particular, it inherits all of the standard *write()* methods provided by *java.io.Writer*, and also implements all of the *print()* and *println()* methods defined by java.io.PrintWriter.

For example the *out* object can be used within a scriptlet to add content to the generated page as in the following page fragment:

```
<p> Counting eggs
<% int count=0;
  while (carton.hasNext()){
          count++;
          out.print(".");
}
%><p>
There are <%=count%> eggs.</p>
```

The javax.servlet.jspWriter class defines a number of methods that supports JSP specific behavior. These additional methods are summarized in following table:

Method	Description
isAutoFlush()	Returns true if the output buffer is automatically flushed when it becomes full, false if an exception is thrown.
getBufferSize()	Returns the size (in bytes) of the output buffer
getRemaining()	Returns the size (in bytes) of the unused portion of the output buffer
clearBuffer()	Clears the contents of the output buffer, discarding them.

clear()	Clears the contents of the output buffer, signaling an error if the buffer has previously been flushed.
newLine()	Writes a (Platform-specific) line separator to the output buffer.
flush()	Flushes the output buffer, then flushes the output stream
close()	Closes the output stream flushing any comments

Table 12. JSP-oriented methods of the javax.servlet.jsp.JspWriter Interface

Below is a page fragment that uses the out object to display the buffering status:

```
<% int total=out.getBufferSize();
     int available = out.getRemaining();
    int used = total – available; %>
Buffering status:
<%=used%>/<%=total%> =<%= (100.0 * used)/TOTAL %>
```

The methods provided for clearing the buffer are also particularly useful. Consider following approach:

```
<% out.flush();
   try{ %>
<P align=center><%=x%>!=<%=fact(x)%></p>
<%} catch(IllegalArgumentException e){
          out.clearBuffer(); %>
<p> Sorry, factorial argument is out of range.</p>
<% } %>
```

In this version, the flush() method is called on the out object to empty the buffer and make sure all of the content generated so far is displayed. Then the try block is opened and the call to the fact method, which has the potential of throwing an IllegalArgumentException, is made.

7.1.4 Contextual objects

The implicit objects in this category provide the JSP page with access to the context within which it is being processed. The *session* object for example, provides the context for the request to which the page is responding. What data has already been associated with the individual user who is requesting the page? The *application* object provides the server-side context within which the page is running.

7.1.4.1 Session object

The JSP implicit object represents an individual user's current session. All of the requests made by a user that are part of a single series of interactions with the web server are considered to be part of a session. As long as new requests by the user continue to be received by the server, the session persists. If a certain length of time passes without any requests from one user, the session expires.

The *session* object then stores information about the session. Application specific data is typically added to the session by means of attributes. Information about the session itself is available through the other methods of the javax.servlet.http.HttpSession interface, of which the session object is an instance.

Method	Description
getId()	Returns the session ID
getCreationTime()	Returns the time at which the session was created
getLastAccessTime()	Returns the last time a request associated with thee session was received
getMaxInactiveInterval()	Returns the maximum time (in seconds) between requests for which thee session will be maintained
setMaxInactiveInterval(t)	Sets the maximum time (in seconds) between requests for which session will be maintained.
isNew()	Returns true if user's browser has not yet confirmed the session ID
invalidate()	Discards the session releasing any objects stored as attributes

Table 13 – Methods of the javax.servlet.http.HttpSession Interface

One of the primary uses for the *session* object is the storing and retrieving of attribute values, in order to transmit user specific information between pages. Below is a scriplet that stores data in the session in the form of a hypothetical UserLogin object:

```
<%UserLogin userData = new UserLogin(name, password);
 session.setAttribute("login", userData); %>
```

Once this scriptlet has been used to store the data via the setAttribute() method, another scripting element – either on the same JSP page or on another page later visited by the user – could access that same data using the getAttribute() method, as in the following:

```
<% UserLogin userData = (UserLogin) session.getAttribute("login");
    if(userData.isGroupMember("admin")){
        session.setMaxInactiveInterval(6-*60*8);
} else{
  session.setMaxInactiveInterval(60*15);
}
%>
```

Note that when this scriplet retrieves the stored data, it must use the casting operator to restore its type. This is because the base type for attribute values is java.lang.Object, which is therefore the return type for the getAttribute() method.

Note that JSP provides a mechanism for objects to be notified when they are added to or removed from a user's session. If an object is stored in a session and its class implements the javax.servlet.http.HttpSessionBindingListener interface, then certain methods required by that interface will be called whenever session-related events occur.

7.1.4.2 Application object

This implicit object represents the application to which the JSP page belongs. It is an instance of the javax.servlet.ServletContext interface. JSP pages are grouped into applications according to their URLs. JSP containers typically treat the first directory name in a URL as an application. For example http://server/games/index.jsp, http://server/games/matrixblaster.jsp and http://server/games/space/paranoids.jsp are all considered part of the same *games* application. Alternatively complete control over application grouping can be obtained by use of Web Application Descriptor files.

The key methods of the javax.servlet.ServletContext interface can be grouped into five major categories:

Methods in Table 14 allow the developer to retrieve version information from the servlet container.

Method	Description
getServerInfo()	Returns the name and version of the servlet container
getMajorVersion()	Returns the major version of the Servlet API for the servlet container
getMinorVersion()	Returns the minor version of the Servlet API for the servlet container

Table 14 Container methods of the javax.servlet.ServletContext interface

Table 15 lists several methods for accessing server-side resources represented as filenames and URLs.

Method	Description
getMimeType(filename)	Returns the MIME type for the indicated file, if known by the server.
getResource(path)	Translates a string specifying a URL into an object that accesses the URL's contents, either locally or over the network.
getResourceAsStream(path)	Translates a string specifying a URL into an input stream for reading its contents.
getRealPath(path)	Translates a local URL into a path name in the local file system
getContext(path)	Returns the application context for the specified local URL
getRequestDispatcher(path)	Creates a request dispatcher for the indicated local URL

Table 15 Methods of the javax.servlet.ServletContext interface for interacting with server-side paths and files

The application object also provides support for logging, via the methods summarized in Table 16

Method	Description
log(message)	Writes the message to the log file
log(message,exception)	Writes the message to the log file, along with the stack trace for the specified exception

Table 16 Methods of the javax.servlet.ServletContext interface for message logging

The fourth set of methods supported by this interface are those for getting and setting attribute values. A final pair of methods provides access to initialization parameters associated with the application as a whole.

7.1.4.3 PageContext object

The pageContext object provides programmatic access to all other implicit objects. For the implicit objects that support attributes, the pageContext object also provides methods for accessing those attributes. In addition, the pageContext object implements methods for transferring control from the current page to another page, either temporarily to generate output to be included in the output of the current page, or permanently to transfer control altogether. The *pageContext* object is an instance of the *javax.servlet.jsp.PageContext* class.

Method	Description
getPage()	Returns the servlet instance for the current page(i.e. the page implicit object)
getRequest()	Returns the request that initiated the processing of the page(i.e. the request implicit object)
getResponse()	Returns the response for the page(i.e. the response implicit object)
getOut()	Returns the current output stream for the page (i.e. the out implicit object).
getSession()	Returns the session associated with the current page request
getServletConfig()	Returns the servlet configuration object(i.e. the config implicit object)
getServletContext()	Returns the context in which the page's servlet runs(i.e. the application implicit object)
getException()	For error pages, returns the exception passed to the page (i.e. the exception

Table 17 Methods of the javax.servlet.jsp.PageContext class for programmatically retrieving the JSP implicit objects

Method	Description
Forward(path)	Forwards processing to another local URL
Include(path)	Includes the output from processing another local URL

Table 18 Request dispatch methods of the javax.servlet.jsp.PageContext class

Two of other groups of methods supported by the pageContext object deal with attributes. This implicit object is among those capable of storing attributes.

Method	Description
setAttribute(key,value,scope)	Associates an attribute value with a key in a specific scope
getAttributeNamesInScope(scope)	Retrieves the names of all attributes in a specific scope
getAttribute(key,scope)	Retrieves the attribute value associated with the key in a specific scope
removeAttribute(key,scope)	Removes the attribute value associated with the key in a specific scope
findAttribute(name)	Searches all scopes for the named attribute
getAttributeScope(name)	Returns the scope in which the named attribute is stored

Table 19 Methods of the javax.servlet.jsp.PageContext class for accessing attributes across multiple scopes

Four different implicit objects are capable of storing attributes: the *pageContext* object, the *request* object, the *session* object and the *application* object. As a result of this ability, these objects are also referred to as *scopes*, because the longevity of an attribute value is a direct result of the four objects in which it is stored. Page attributes, stored in the pageContext object, only last as long as the processing of a single page. Request attributes are also short-lived, but may be passed between pages as control is transferred. Session attributes persist as long as the user continues interacting with the web server. Application attributes are retained as long as the JSP container keeps one or more of an application's pages loaded in memory – conceivably, as long as the JSP container is running.

In conjunction with the methods listed in Table 20 whose parameters include a scope specification, the javax.servlet.jsp.PageContext class provides static variables for representing these four different scopes. Behind the scenes, these are just symbolic names for four arbitrary integer values. Since the actual values are hidden though, the symbolic names are the standard means for indicating attribute scopes, as in the following page fragment:

```
<%@page import="javax.servlet.jsp.PageContext" %>
<% Enumeration stts=
pageContext.getAttributeNamesInScope(PageContext.SESSION_SCOPE);
while (atts.hasMoreElements()){ %>
Session Attribute: <%= atts.nextElement() %> <br>
<% } %>
```

Variable	Description
PAGE_SCOPE	Scope for attributes stored in the pageContext object
REQUEST_SCOPE	Scope for attributes stored in the request object
SESSION_SCOPE	Scope for attributes stored in the session object

APPLICATION_SCOPE	Scope for attributes stored in the application object

Table 20 Class scope variables for the javax.servlet.jsp.PageContext class

7.1.4.4 Exception object

The ninth and final JSP implicit object is the *exception* object. Like the *session* object, the *exception* object is not automatically available on every JSP page. Instead this object is only available on pages that have been designated as error page using the *isErrorPage* attribute of the page directive. On those JSP pages that are error pages, the exception object will be an instance of the java.lang.Throwable class corresponding to the uncaught error that caused control to be transferred to the error page. The methods of the java.lang.Throwable class that are particularly useful in the context of JSP are summarized in Table 21

Method	Description
getMessage()	Returns the descriptive error message associated with the exception
printStackTrace(out)	Prints the execution stack in effect when the exception was thrown to the designated output stream
toString()	Returns a string combining the class name of the exception with its error message

Table 21 Relevant methods of the java.lang.Throwable class

Here is an example page fragment demonstrating the use of exception object:

<%@ page isErrorPage=”true” %>
<H1>Warning!</H1>
The following error has been detected:
<B>**<%=exeption%>**</B>
<% exception.printStackTrace(out); %>

8 Session Tracking in JSP (Session Object)

- ***Cookies*** – a small text file stored on the client’s machine. Cookies can be disabled in the browser settings so are not always available.
- ***URL rewriting*** – store session information in the URL. Works when cookies are not supported but can make bookmaking of web pages a problem because they have session specific information at the end of a URL.
- ***Hidden form fields*** – HTML hidden edit boxes such as <INPUT TYPE=”HIDDEN” NAME=”USERNAME” VALUE=”..’”>. Every page has to be dynamically produced with the values in the hidden field.
- ***Session objects*** – JSP Implicit object

A session object uses a key/value combination to store information. To retrieve information from a session:

Session.getValue(“counter”);

The return type of the method getValue is Object, so we need to typecast to get the required value. If there is not a session key with that name, a null is returned.

To set a session key with a value,

Session.putValue(“counter”,totalCount)

<%-- JSP comment --%>

8.1 An example about JSP session:

```
<!-- session.jsp
checks to see if you have visited a page and keeps a counter. www.porasl.com
--->

<html>
<head>
</head>
<body>
<%
// get the value of the session variable - visitcounter
  Integer totalvisits = (Integer) session.getValue("visitcounter");

// if the session variable (visitcounter) is null
    if (totalvisits == null)
    {
  //set session variable to 0
    totalvisits = new Integer(0);
    session.putValue("visitcounter",totalvisits);

   // print a message to out visitor
   out.println("Welcome,visitor");
  }
  else
  {
   // if you have visited the page before then add 1 to the visitcounter
  totalvisits = new Integer(totalvisits.intValue()+1);
  session.putValue("visitcounter",totalvisits);
  out.println("You have visited this page "+ totalvisits + " time(s)!");
}
%>
</body>
</html>
```

9 Flow of control

This ability of scriptlets to introduce statement blocks without closing them can be put to good use in JSP pages to affect the flow of control through the various elements, static or dynamic, that govern page output. In particular, such scriptlets can be used to implement conditional or iterative content, or to add error handling to a sequence of operations.

9.1 Conditionalization

Java's if statement with optional *else if* and *else* clauses, is used to control the execution of code based on logical true/false tests.

```
<% if (x<0) {%>
<p>Sorry, can't compute the factorial of a negative number.</p>
<%} else if (x>20){%>
<p>Sorry, arguments greater than 20 cause an overflow error.</p>
<%} else{%>
<p align=center><% =x%> !=<% =fact(x) %></p>
<%}%>
```

Three different blocks of statements are created by these scriptlets, only one of which will actually be executed.

9.2 Iteration

Java has three different iteration constructs: the for loop, the while loop and the do/while statement. They may all be used via scriplets to add iterative content to a JSP page, and are particularly useful in the display of tabular data.

```
<table>
<tr><th><I>X</I></th><th><I>X</I> !</th></tr>
<% for (long x=01; x<=201; ++X){%>
<tr><td><%  -x%></td><td><%=fact(x) %></td></tr>
<% }%>
</table>
```

Following is a short example of a JSP code illustrating above-mentioned iteration:

```
<html>
<head><title>
</title>
</head>
<body>
   <%! public long fact (long x) throws IllegalArgumentException{
        if((x<0) || (x>20))
                throw new IllegalArgumentException("Out of range.");
        else if(x==0) return 1;
        else return x*fact(x-1);
} %>
```

```
<table border="1">
<tr><th><I>X</I></th><th><I>X</I> !</th></tr>
<% for (long x=1; x<=7; ++x){%>
<tr><td><%=x%></td><td><%=fact(x)%></td></tr>
<% }%>
</table>
</body>
</html>
```

Output of this JSP is following HTML file:

X	*X* !
1	1
2	2
3	6
4	24
5	120
6	720
7	5040

Table 22 view of Generated HTML file

9.3 Exception handling

The default behavior when a exception is thrown while processing a JSP page is to display an implementation specific error message in the browser window. If a block of code on a JSP page has the potential of signaling an error, java's exception handling construct, the try block, maybe used in a set of scriptlets to catch the error locally and respond to it gracefully within the current page. Consider following alternative declaration for the factorial method presented earlier:

```
<%! Public long fact (long x) throws IllegalArgumentException{
        if((x<0) || (x>20))
                throw new IllegalArgumentException("Out of range.");
        else if(x==0) return 1;
        else return x*fact(x-1);
} %>
```

This version of the method verifies that the method's argument is within the valid range for this calculation, signaling an IllegalArgumentException if it is not. Using this version of the method, we could consider an alternative implementation of the example presented in the foregoing section on conditionals, as follows:

```
<%try {%>
<p align=center><%=x%> !=<%=fact(x)%></p>
```

```
<%} catch(IllegalArgumentException e){%>
<p> Sorry factorial argument is out of range.</p>
<%}%>
```

9.4 Comments

9.4.1 Content comments

Content comment's syntax associated with the type of content being generated by the JSP page. To write a comment that will be included in the output of a JSP page that is generating web content, the following syntax is used:

```
 <!- -comment - ->
```

Since these comments are part of the output from the page, we can include dynamic content in them. HTML and XML comments can include JSP expressions, and the output generated by these expressions will appear as part of the comment in the page's response. For example:

```
<!- - Java longs are 64 bits, so 20! =<%=fact(20)%> is the upper limit - ->
```

In this case, the computed value of the factorial expression will appear in the comment that is actually sent to the browser.

9.4.2 JSP comments

JSP comments are independent of the type of content being produced by the page. They are also independent of the scripting language used by the page. These comments can only be viewed by examining the original JSP file and take the following form:

```
<%-- comment --%>
```

The body of this comment is ignored by the JSP container. When the page is compiled into a servlet anything appearing between these two delimiters is skipped while translating the page into servlet source code.

In the following page fragment only the first and last expressions displaying the factorials of 2 and 6, will appear in the page output:

```
2! =<%= fact(2)%><br>
<%--
3! =<%= fact(3)%><br>
4! =<%= fact(4)%><br>
5! =<%= fact(5)%><br>
-- >
6! =<%= fact(6)%>
```

All of the other expressions have been commented out and will not appear in the page's output.

9.4.3 Scripting language comments

Comments may also be introduced into a JSP page within scriptlets using the native comment syntax of scripting language. For example java uses /* and */ as comment delimiters. With java as the JSP scripting language, then scripting language comments take the following form:

```
<% /* Script language comments */%>
```

Like JSP comments scripting language comments will not appear in the page's output.

9.4.4 Example for Comments in JSP

When we run following JSP code:

```
<html>
<head><title>
</title>
</head>
<body>
    <%! public long fact (long x) throws IllegalArgumentException{
          if((x<0) || (x>20))
                throw new IllegalArgumentException("Out of range.");
          else if(x==0) return 1;
          else return x*fact(x-1);
} %>

<!-- 1. Content comments will be sent to client-->
<br>
 2!=<%=fact(2)%><br>
<%-- 2. JSP comment will be ignored by servlet engine ONLY 2! and 6! will be
displayed in generated HTML file.
3!=<%=fact(3)%><br>
 4!=<%=fact(4)%><br>
 5!=<%=fact(5)%><br>
--%>
 6!=<%=fact(6)%><br>

<% /* Script language comments */%>
</body>
</html>
```

Servlet engine generates following HTML code:

```
<html>
<head><title>
</title>
```

```
</head>
<body>

<!-- 1. Content comments will be sent to client-->
<br>
 2!=2<br>
 6!=720<br>
</body>
</html>
```

In above generated HTML code comments number 2 and 3, i.e. “JSP comments” and “script language comments” are removed by servlet compiler.

10 Actions

Actions are the fourth and final major category of JSP tags and themselves serve three major roles:

- JSP actions allow for the transfer of control between pages.
- Actions support the specification of Java applets in a browser-independent manner
- Action enable JSP pages to interact with JavaBeans component objects residing on the server

All custom tags defined via tag libraries take the from of JSP action.

10.1 Forward

The <jsp:forward> action is used to permanently transfer control from a JSP page to another location in the local server. Any content generated by the current page is discarded, and processing of the request begins a new at the alternate location. The basic syntax for this JSP action is as follows:

<jsp:forward page=”localURL” />

The page attribute of the <jsp:forward> action is used to specify this alternate location to which control should be transferred, which may be a static document, a CGI, a servlet or another JSP page. Note that the browser from, which the request was submitted isn’t notified when the request is transferred to this alternate URL. In particular, the location field at the top of the browser window will continue to display the URL that was originally requested.

For added flexibility, the <jsp:forward> action supports the use of request-time attribute values for page attribute. Specifically this means that a JSP expression can be used to specify the value of the page attributes, as in the following example:

<jsp:forward page=’<%= “message” + statusCode+”.html” %>’ />

Every time the page is processed for a request and the <jsp:forward action is to be taken, this expression will be evaluated by the JSP container, and the resulting value will be interpreted as the URL to which the request should be forwarded.

As mentioned above <jsp:forward> action can be used to transfer control to any other document on the local serevr. For the specific case when control is transferred to another JSP page the JSP container will automatically assign a new *pageContext* object to the forwarded page. The *request* object and the *session* object will be the same for both the original page and the forwarded page. Sharing *application* object depends upon whether or not the two pages are both part of the same application. As a result some, but not all, of attribute values accessible from the original page will be accessible on the forwarded page, depending upon their scope: page attributes are not shared.

If you need to transfer data as well as control from one page to another, the typical approach is to store this data either in the request or in the session, depending upon how much longer the data will be needed. All of the objects in which JSP pages can store attributes are also accessible via the servlet API. As a result, this approach can also be used to transfer data when forwarding from JSP page to a servlet.

Since the request object is common to both the original page and the forwarded page, any request parameters that were available on the original page will also be accessible from the forwarded page. It is also possible to specify additional request parameters to be sent to the forwarded page through use of the <jsp:param> tag within the body of the <jsp:forward> action.

10.1.1 Example - Jsp:forward

Sending additional parameters to forwarded page:

```
<jsp:forward page="localURL">
  <jsp:param name="paramName1"  value="paramValue1">
 …..
 <jsp:param name="paramName1"  value="paramValue1">
</jsp:forward>
```

Forwarding a page:

```
<% if(! Database.isAvailable()) { %>
  <% Notify the user about routine maintenance. %>
  <jsp:forward  page="db-maintenance.html" />
<% } %>
<% - - Database is up and running …. - -%>
```

10.2 Include

The <jsp:include> action enables page authors to incorporate the content generated by another local document into the output of the current page. The output from the included

Document is inserted into the original page's output in place of the <jsp:include> tag, then, this action is used to *temporarily* transfer control from a JSP page to another location on the local server.

<jsp:include page="localURL" flush="true" />

The page attribute of the <jsp:include> action is used to identify the document whose output is to be inserted into the current page, and is specified as a URL on the local server. The included page can be a static document, a CGI a servlet, or another JSP page. As with the <jsp:forward> action, the page attribute of the <jsp:include> action supports request-time attribute values (i.e., specifying is value via a JSP exprestion).

The *flush* attribute of the <jsp:include> action controls whether or not the output buffer for the current page is flushed prior to including the content from the included page. It is required that the *flush* attribute be set to true, indicating that the buffer is flushed before processing of the included page begins.

Another element of functionality that the <jsp:include> action has in common with the <jsp:forward> action is the ability to specify additional request parameters for the included document. Again, this is accomplished via use of the <jsp:param> tag within the body of the <jsp:include> action, as follows:

```
<jsp:include page="localURL" flush="true">
  <jsp:include name="paramName1"  value="paramValue1" />
<jsp:include name="paramName2"  value="paramValue2" />
</jsp:include>
```

As indicated in figure the <jsp:include> action works by passing its request on the included page, which is then handled by the JSP container as it would be handled any other request. The output from the included page is then folded into the output of the original page, which resumes processing. This incorporation of content takes place at the time the request is handled. In addition, because the JSP container automatically generates and compiles new servlets for JSP pages that have charged, if the text in a JSP file included via the <jsp:include> action is changed, the changes will automatically be reflected in the output of the including file. When the request is directed from the original file to the included JSP page, the standard JSP mechanisms – that is, translation into a stand-alone servlet, with automatic recompilation of changed files – are employed to process the included page.

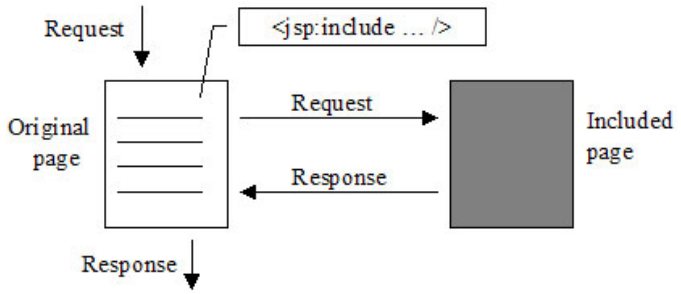

Figure 1 <jsp:include> action on the processing of a request

The JSP include directive doesn’t automatically update the including page when the included file is modified. This is because the *include* directive takes effect when the including page istranslated into a servlet, effectively merging the base contents of the included page into those of the original. The <jsp:include> action takes effect when processing requests, and merges the output from the included page rather than its original text.

10.3 Plug-in

The <jsp:plugin> action is used to generate browser-specific HTML for specifying Java applets which reply on Sun Microsystem’s java plug-in.

10.4 Bean tags

JSP provides three different actions for interacting ith server-side JavaBeans: <jsp:UseBean>, <jsp:setProperty> and <jsp:getProperty>. Because component-centric design provides key strengths with respect to separation of presentation and application logic. Interaction between JSP and JavaBean is described in this document.