The Scriptlet:

A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language.

Following is the syntax of Scriptlet:

<% code fragment %>

You can write XML equivalent of the above syntax as follows:

<jsp:scriptlet>

code fragment

</jsp:scriptlet>

Any text, HTML tags, or JSP elements you write must be outside the scriptlet. Following is the simple and first example for JSP:

<html>

<head><title>Hello World</title></head>

<body>

Hello World!<br/>

<%

out.println("Your IP address is " + request.getRemoteAddr());

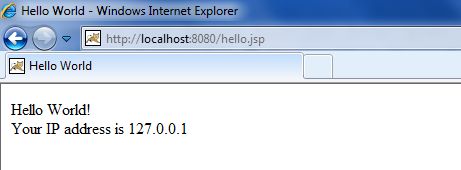
%>

</body>

</html>

**NOTE:** Assuming that Apache Tomcat is installed in C:\apache-tomcat-7.0.2 and your environment is setup as per environment setup tutorial.

Let us keep above code in JSP file hello.jsp and put this file in **C:\apache-tomcat-7.0.2\webapps\ROOT** directory and try to browse it by giving URL http://localhost:8080/hello.jsp. This would generate following result:



JSP Declarations:

A declaration declares one or more variables or methods that you can use in Java code later in the JSP file. You must declare the variable or method before you use it in the JSP file.

Following is the syntax of JSP Declarations:

<%! declaration; [ declaration; ]+ ... %>

You can write XML equivalent of the above syntax as follows:

<jsp:declaration>

code fragment

</jsp:declaration>

Following is the simple example for JSP Declarations:

<%! int i = 0; %>

<%! int a, b, c; %>

<%! Circle a = new Circle(2.0); %>

JSP Expression:

A JSP expression element contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file.

Because the value of an expression is converted to a String, you can use an expression within a line of text, whether or not it is tagged with HTML, in a JSP file.

The expression element can contain any expression that is valid according to the Java Language Specification but you cannot use a semicolon to end an expression.

Following is the syntax of JSP Expression:

<%= expression %>

You can write XML equivalent of the above syntax as follows:

<jsp:expression>

expression

</jsp:expression>

Following is the simple example for JSP Expression:

<html>

<head><title>A Comment Test</title></head>

<body>

<p>

Today's date: <%= (new java.util.Date()).toLocaleString()%>

</p>

</body>

</html>

This would generate following result:

|  |
| --- |
| Today's date: 11-Sep-2010 21:24:25 |

JSP Comments:

JSP comment marks text or statements that the JSP container should ignore. A JSP comment is useful when you want to hide or "comment out" part of your JSP page.

Following is the syntax of JSP comments:

<%-- This is JSP comment --%>

Following is the simple example for JSP Comments:

<html>

<head><title>A Comment Test</title></head>

<body>

<h2>A Test of Comments</h2>

<%-- This comment will not be visible in the page source --%>

</body>

</html>

This would generate following result:

|  |
| --- |
| A Test of Comments |

There are a small number of special constructs you can use in various cases to insert comments or characters that would otherwise be treated specially. Here's a summary:

|  |  |
| --- | --- |
| **Syntax** | **Purpose** |
| <%-- comment --%> | A JSP comment. Ignored by the JSP engine. |
| <!-- comment --> | An HTML comment. Ignored by the browser. |
| <\% | Represents static <% literal. |
| %\> | Represents static %> literal. |
| \' | A single quote in an attribute that uses single quotes. |
| \" | A double quote in an attribute that uses double quotes. |

JSP Directives:

A JSP directive affects the overall structure of the servlet class. It usually has the following form:

<%@ directive attribute="value" %>

There are three types of directive tag:

|  |  |
| --- | --- |
| **Directive** | **Description** |
| <%@ page ... %> | Defines page-dependent attributes, such as scripting language, error page, and buffering requirements. |
| <%@ include ... %> | Includes a file during the translation phase. |
| <%@ taglib ... %> | Declares a tag library, containing custom actions, used in the page |

We would explain JSP directive in separate chapter [JSP - Directives](http://www.tutorialspoint.com/jsp/jsp_directives.htm)

JSP Actions:

JSP actions use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.

There is only one syntax for the Action element, as it conforms to the XML standard:

<jsp:action\_name attribute="value" />

Action elements are basically predefined functions and there are following JSP actions available:

|  |  |
| --- | --- |
| **Syntax** | **Purpose** |
| jsp:include | Includes a file at the time the page is requested |
| jsp:useBean | Finds or instantiates a JavaBean |
| jsp:setProperty | Sets the property of a JavaBean |
| jsp:getProperty | Inserts the property of a JavaBean into the output |
| jsp:forward | Forwards the requester to a new page |
| jsp:plugin | Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin |
| jsp:element | Defines XML elements dynamically. |
| jsp:attribute | Defines dynamically defined XML element's attribute. |
| jsp:body | Defines dynamically defined XML element's body. |
| jsp:text | Use to write template text in JSP pages and documents. |

We would explain JSP actions in separate chapter [JSP - Actions](http://www.tutorialspoint.com/jsp/jsp_actions.htm)

JSP Implicit Objects:

JSP supports nine automatically defined variables, which are also called implicit objects. These variables are:

|  |  |
| --- | --- |
| **Objects** | **Description** |
| request | This is the **HttpServletRequest** object associated with the request. |
| response | This is the **HttpServletResponse** object associated with the response to the client. |
| out | This is the **PrintWriter** object used to send output to the client. |
| session | This is the **HttpSession** object associated with the request. |
| application | This is the **ServletContext** object associated with application context. |
| config | This is the **ServletConfig** object associated with the page. |
| pageContext | This encapsulates use of server-specific features like higher performance **JspWriters**. |
| page | This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class. |
| Exception | The **Exception** object allows the exception data to be accessed by designated JSP. |

We would explain JSP Implicit Objects in separate chapter [JSP - Implicit Objects](http://www.tutorialspoint.com/jsp/jsp_implicit_objects.htm).

Control-Flow Statements:

JSP provides full power of Java to be embedded in your web application. You can use all the APIs and building blocks of Java in your JSP programming including decision making statements, loops etc.

Decision-Making Statements:

The **if...else** block starts out like an ordinary Scriptlet, but the Scriptlet is closed at each line with HTML text included between Scriptlet tags.

<%! int day = 3; %>

<html>

<head><title>IF...ELSE Example</title></head>

<body>

<% if (day == 1 | day == 7) { %>

<p> Today is weekend</p>

<% } else { %>

<p> Today is not weekend</p>

<% } %>

</body>

</html>

This would produce following result:

|  |
| --- |
| Today is not weekend |

Now look at the following **switch...case** block which has been written a bit differentlty using**out.println()** and inside Scriptletas:

<%! int day = 3; %>

<html>

<head><title>SWITCH...CASE Example</title></head>

<body>

<%

switch(day) {

case 0:

out.println("It\'s Sunday.");

break;

case 1:

out.println("It\'s Monday.");

break;

case 2:

out.println("It\'s Tuesday.");

break;

case 3:

out.println("It\'s Wednesday.");

break;

case 4:

out.println("It\'s Thursday.");

break;

case 5:

out.println("It\'s Friday.");

break;

default:

out.println("It's Saturday.");

}

%>

</body>

</html>

This would produce following result:

|  |
| --- |
| It's Wednesday. |

Loop Statements:

You can also use three basic types of looping blocks in Java: **for, while,and do…while** blocks in your JSP programming.

Let us look at the following **for** loop example:

<%! int fontSize; %>

<html>

<head><title>FOR LOOP Example</title></head>

<body>

<%for ( fontSize = 1; fontSize <= 3; fontSize++){ %>

<font color="green" size="<%= fontSize %>">

JSP Tutorial

</font><br />

<%}%>

</body>

</html>

This would produce following result:

|  |
| --- |
| JSP Tutorial  JSP Tutorial  JSP Tutorial |

Above example can be written using **while** loop as follows:

<%! int fontSize; %>

<html>

<head><title>WHILE LOOP Example</title></head>

<body>

<%while ( fontSize <= 3){ %>

<font color="green" size="<%= fontSize %>">

JSP Tutorial

</font><br />

<%fontSize++;%>

<%}%>

</body>

</html>

This would also produce following result:

|  |
| --- |
| JSP Tutorial  JSP Tutorial  JSP Tutorial |

JSP Operators:

JSP supports all the logical and arithmetic operators supported by Java. Following table give a list of all the operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

Within an expression, higher precedence operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] . (dot operator) | Left to right |
| Unary | ++ - - ! ~ | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | >> >>> << | Left to right |
| Relational | > >= < <= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %= >>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

JSP Literals:

The JSP expression language defines the following literals:

* **Boolean:** true and false
* **Integer:** as in Java
* **Floating point:** as in Java
* **String:** with single and double quotes; " is escaped as \", ' is escaped as \', and \ is escaped as \\.
* **Null:** null
* JSP directives provide directions and instructions to the container, telling it how to handle certain aspects of JSP processing.
* A JSP directive affects the overall structure of the servlet class. It usually has the following form:
* <%@ directive attribute="value" %>
* Directives can have a number of attributes which you can list down as key-value pairs and separated by commas.
* The blanks between the @ symbol and the directive name, and between the last attribute and the closing %>, are optional.
* There are three types of directive tag:

|  |  |
| --- | --- |
| **Directive** | **Description** |
| <%@ page ... %> | Defines page-dependent attributes, such as scripting language, error page, and buffering requirements. |
| <%@ include ... %> | Includes a file during the translation phase. |
| <%@ taglib ... %> | Declares a tag library, containing custom actions, used in the page |

## The page Directive:

* The **page** directive is used to provide instructions to the container that pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.
* Following is the basic syntax of page directive:
* <%@ page attribute="value" %>
* You can write XML equivalent of the above syntax as follows:
* <jsp:directive.page attribute="value" />

## Attributes:

* Following is the list of attributes associated with page directive:

|  |  |
| --- | --- |
| **Attribute** | **Purpose** |
| buffer | Specifies a buffering model for the output stream. |
| autoFlush | Controls the behavior of the servlet output buffer. |
| contentType | Defines the character encoding scheme. |
| errorPage | Defines the URL of another JSP that reports on Java unchecked runtime exceptions. |
| isErrorPage | Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute. |
| extends | Specifies a superclass that the generated servlet must extend |
| import | Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes. |
| info | Defines a string that can be accessed with the servlet's getServletInfo() method. |
| isThreadSafe | Defines the threading model for the generated servlet. |
| language | Defines the programming language used in the JSP page. |
| session | Specifies whether or not the JSP page participates in HTTP sessions |
| isELIgnored | Specifies whether or not EL expression within the JSP page will be ignored. |
| isScriptingEnabled | Determines if scripting elements are allowed for use. |

* Check more detail related to all the above attributes at [Page Directive](http://www.tutorialspoint.com/jsp/page_directive.htm).

## The include Directive:

* The **include** directive is used to includes a file during the translation phase. This directive tells the container to merge the content of other external files with the current JSP during the translation phase. You may code *include* directives anywhere in your JSP page.
* The general usage form of this directive is as follows:
* <%@ include file="relative url" >
* The filename in the include directive is actually a relative URL. If you just specify a filename with no associated path, the JSP compiler assumes that the file is in the same directory as your JSP.
* You can write XML equivalent of the above syntax as follows:
* <jsp:directive.include file="relative url" />
* Check more detail related to include directive at [Include Directive](http://www.tutorialspoint.com/jsp/include_directive.htm).

## The taglib Directive:

* The JavaServer Pages API allows you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior.
* The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides a means for identifying the custom tags in your JSP page.
* The taglib directive follows the following syntax:
* <%@ taglib uri="uri" prefix="prefixOfTag" >
* Where the **uri** attribute value resolves to a location the container understands and the **prefix** attribute informs a container what bits of markup are custom actions.
* You can write XML equivalent of the above syntax as follows:
* <jsp:directive.taglib uri="uri" prefix="prefixOfTag" />
* Check more detail related to taglib directive at [Taglib Directive](http://www.tutorialspoint.com/jsp/taglib_directive.htm).

JSP actions use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.

There is only one syntax for the Action element, as it conforms to the XML standard:

<jsp:action\_name attribute="value" />

Action elements are basically predefined functions and there are following JSP actions available:

|  |  |
| --- | --- |
| **Syntax** | **Purpose** |
| jsp:include | Includes a file at the time the page is requested |
| jsp:useBean | Finds or instantiates a JavaBean |
| jsp:setProperty | Sets the property of a JavaBean |
| jsp:getProperty | Inserts the property of a JavaBean into the output |
| jsp:forward | Forwards the requester to a new page |
| jsp:plugin | Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin |
| jsp:element | Defines XML elements dynamically. |
| jsp:attribute | Defines dynamically defined XML element's attribute. |
| jsp:body | Defines dynamically defined XML element's body. |
|  |  |
| jsp:text | Use to write template text in JSP pages and documents. |

Common Attributes:

There are two attributes that are common to all Action elements: the **id** attribute and the **scope** attribute.

* **Id attribute:** The id attribute uniquely identifies the Action element, and allows the action to be referenced inside the JSP page. If the Action creates an instance of an object the id value can be used to reference it through the implicit object PageContext
* **Scope attribute:** This attribute identifies the lifecycle of the Action element. The id attribute and the scope attribute are directly related, as the scope attribute determines the lifespan of the object associated with the id. The scope attribute has four possible values: (a) page, (b)request, (c)session, and (d) application.

The <jsp:include> Action

This action lets you insert files into the page being generated. The syntax looks like this:

<jsp:include page="relative URL" flush="true" />

Unlike the **include** directive, which inserts the file at the time the JSP page is translated into a servlet, this action inserts the file at the time the page is requested.

Following is the list of attributes associated with include action:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| page | The relative URL of the page to be included. |
| flush | The boolean attribute determines whether the included resource has its buffer flushed before it is included. |

Example:

Let us define following two files (a)date.jps and (b) main.jsp as follows:

Following is the content of date.jsp file:

<p>

Today's date: <%= (new java.util.Date()).toLocaleString()%>

</p>

Here is the content of main.jsp file:

<html>

<head>

<title>The include Action Example</title>

</head>

<body>

<center>

<h2>The include action Example</h2>

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

</center>

</body>

</html>

Now let us keep all these files in root directory and try to access main.jsp. This would display result something like this:

|  |
| --- |
| The include action Example  Today's date: 12-Sep-2010 14:54:22 |

The <jsp:useBean> Action

The **useBean** action is quite versatile. It first searches for an existing object utilizing the id and scope variables. If an object is not found, it then tries to create the specified object.

The simplest way to load a bean is as follows:

<jsp:useBean id="name" class="package.class" />

Once a bean class is loaded, you can use **jsp:setProperty** and **jsp:getProperty** actions to modify and retrieve bean properties.

Following is the list of attributes associated with useBean action:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| Class | Designates the full package name of the bean. |
| Type | Specifies the type of the variable that will refer to the object. |
| beanName | Gives the name of the bean as specified by the instantiate () method of the java.beans.Beans class. |

Let us discuss about **jsp:setProperty** and **jsp:getProperty** actions before giving a valid example related to these actions.

The <jsp:setProperty> Action

The **setProperty** action sets the properties of a Bean. The Bean must have been previously defined before this action. There are two basic ways to use the setProperty action:

You can use jsp:setProperty after, but outside of, a jsp:useBean element, as below:

<jsp:useBean id="myName" ... />

...

<jsp:setProperty name="myName" property="someProperty" .../>

In this case, the jsp:setProperty is executed regardless of whether a new bean was instantiated or an existing bean was found.

A second context in which jsp:setProperty can appear is inside the body of a jsp:useBean element, as below:

<jsp:useBean id="myName" ... >

...

<jsp:setProperty name="myName" property="someProperty" .../>

</jsp:useBean>

Here, the jsp:setProperty is executed only if a new object was instantiated, not if an existing one was found.

Following is the list of attributes associated with setProperty action:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| name | Designates the bean whose property will be set. The Bean must have been previously defined. |
| property | Indicates the property you want to set. A value of "\*" means that all request parameters whose names match bean property names will be passed to the appropriate setter methods. |
| value | The value that is to be assigned to the given property. The the parameter's value is null, or the parameter does not exist, the setProperty action is ignored. |
| param | The param attribute is the name of the request parameter whose value the property is to receive. You can't use both value and param, but it is permissible to use neither. |

The <jsp:getProperty> Action

The **getProperty** action is used to retrieve the value of a given property and converts it to a string, and finally inserts it into the output.

The getProperty action has only two attributes, both of which are required ans simple syntax is as follows:

<jsp:useBean id="myName" ... />

...

<jsp:getProperty name="myName" property="someProperty" .../>

Following is the list of required attributes associated with setProperty action:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| name | The name of the Bean that has a property to be retrieved. The Bean must have been previously defined. |
| property | The property attribute is the name of the Bean property to be retrieved. |

Example:

Let us define a test bean which we will use in our example:

/\* File: TestBean.java \*/

package action;

public class TestBean {

private String message = "No message specified";

public String getMessage() {

return(message);

}

public void setMessage(String message) {

this.message = message;

}

}

Compile above code to generated TestBean.class file and make sure that you copied TestBean.class in C:\apache-tomcat-7.0.2\webapps\WEB-INF\classes\action folder and CLASSPATH variable should also be set to this folder:

Now use the following code in main.jsp file which loads the bean and sets/gets a simple String parameter:

<html>

<head>

<title>Using JavaBeans in JSP</title>

</head>

<body>

<center>

<h2>Using JavaBeans in JSP</h2>

<jsp:useBean id="test" class="action.TestBean" />

<jsp:setProperty name="test"

property="message"

value="Hello JSP..." />

<p>Got message....</p>

<jsp:getProperty name="test" property="message" />

</center>

</body>

</html>

Now try to access main.jsp, it would display following result:

|  |
| --- |
| Using JavaBeans in JSP  Got message.... Hello JSP... |

The <jsp:forward> Action

The **forward** action terminates the action of the current page and forwards the request to another resource such as a static page, another JSP page, or a Java Servlet.

The simple syntax of this action is as follows:

<jsp:forward page="Relative URL" />

Following is the list of required attributes associated with forward action:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| page | Should consist of a relative URL of another resource such as a static page, another JSP page, or a Java Servlet. |

Example:

Let us reuse following two files (a) date.jps and (b) main.jsp as follows:

Following is the content of date.jsp file:

<p>

Today's date: <%= (new java.util.Date()).toLocaleString()%>

</p>

Here is the content of main.jsp file:

<html>

<head>

<title>The include Action Example</title>

</head>

<body>

<center>

<h2>The include action Example</h2>

<jsp:forward page="date.jsp" />

</center>

</body>

</html>

Now let us keep all these files in root directory and try to access main.jsp. This would display result something like as below. Here it discarded content from main page and displayed content from forwarded page only.

|  |
| --- |
| Today's date: 12-Sep-2010 14:54:22 |

The <jsp:plugin> Action

The **plugin** action is used to insert Java components into a JSP page. It determines the type of browser and inserts the <object> or <embed> tags as needed.

If the needed plugin is not present, it downloads the plugin and then executes the Java component. The Java component can be either an Applet or a JavaBean.

The plugin action has several attributes that correspond to common HTML tags used to format Java components. The <param> element can also be used to send parameters to the Applet or Bean.

Following is the typical syntax of using plugin action:

<jsp:plugin type="applet" codebase="dirname" code="MyApplet.class"

width="60" height="80">

<jsp:param name="fontcolor" value="red" />

<jsp:param name="background" value="black" />

<jsp:fallback>

Unable to initialize Java Plugin

</jsp:fallback>

</jsp:plugin>

You can try this action using some applet if you are interested. A new element, the <fallback> element, can be used to specify an error string to be sent to the user in case the component fails.

The <jsp:element> Action

The <jsp:attribute> Action

The <jsp:body> Action

The <jsp:element>, lt;jsp:attribute> and <jsp:body> actions are used to define XML elements dynamically. The word dynamically is important, because it means that the XML elements can be generated at request time rather than statically at compile time.

Following is a simple example to define XML elements dynamically:

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

<html xmlns="http://www.w3c.org/1999/xhtml"

xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>

<body>

<jsp:element name="xmlElement">

<jsp:attribute name="xmlElementAttr">

Value for the attribute

</jsp:attribute>

<jsp:body>

Body for XML element

</jsp:body>

</jsp:element>

</body>

</html>

This would produce following HTML code at run time:

<html xmlns="http://www.w3c.org/1999/xhtml"

xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>

<body>

<xmlElement xmlElementAttr="Value for the attribute">

Body for XML element

</xmlElement>

</body>

</html>

The <jsp:text> Action

The <jsp:text> action can be used to write template text in JSP pages and documents. Following is the simple syntax for this action:

<jsp:text>Template data</jsp:text>

The body fo the template cannot contain other elements; it can only contain text and EL expressions ( Note: EL expressions are explained in subsequent chapter). Note that in XML files, you cannot use expressions such as ${whatever > 0}, because the greater than signs are illegal. Instead, use the gt form, such as ${whatever gt 0} or an alternative is to embed the value in a CDATA section.

<jsp:text><![CDATA[<br>]]></jsp:text>

If you need to include a DOCTYPE declaration, for instance for XHTML, you must also use the <jsp:text> element as follows:

<jsp:text><![CDATA[<!DOCTYPE html

PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"DTD/xhtml1-strict.dtd">]]>

</jsp:text>

<head><title>jsp:text action</title></head>

<body>

<books><book><jsp:text>

Welcome to JSP Programming

</jsp:text></book></books>

</body>

</html>

Try above example with and without <jsp:text> action.

When a browser requests for a web page, it sends lot of information to the web server which can not be read directly because this information travel as a part of header of HTTP request. You can check[HTTP Protocol](http://www.tutorialspoint.com/http/index.htm) for more information on this.

Following is the important header information which comes from browser side and you would use very frequently in web programming:

|  |  |
| --- | --- |
| **Header** | **Description** |
| Accept | This header specifies the MIME types that the browser or other clients can handle. Values of **image/png** or **image/jpeg** are the two most common possibilities. |
| Accept-Charset | This header specifies the character sets the browser can use to display the information. For example ISO-8859-1. |
| Accept-Encoding | This header specifies the types of encodings that the browser knows how to handle. Values of **gzip** or **compress** are the two most common possibilities. |
| Accept-Language | This header specifies the client's preferred languages in case the servlet can produce results in more than one language. For example en, en-us, ru, etc. |
| Authorization | This header is used by clients to identify themselves when accessing password-protected Web pages. |
| Connection | This header indicates whether the client can handle persistent HTTP connections. Persistent connections permit the client or other browser to retrieve multiple files with a single request. A value of **Keep-Alive**means that persistent connections should be used |
| Content-Length | This header is applicable only to POST requests and gives the size of the POST data in bytes. |
| Cookie | This header returns cookies to servers that previously sent them to the browser. |
| Host | This header specifies the host and port as given in the original URL. |
| If-Modified-Since | This header indicates that the client wants the page only if it has been changed after the specified date. The server sends a code, 304 which means **Not Modified** header if no newer result is available. |
| If-Unmodified-Since | This header is the reverse of If-Modified-Since; it specifies that the operation should succeed only if the document is older than the specified date. |
| Referer | This header indicates the URL of the referring Web page. For example, if you are at Web page 1 and click on a link to Web page 2, the URL of Web page 1 is included in the Referer header when the browser requests Web page 2. |
| User-Agent | This header identifies the browser or other client making the request and can be used to return different content to different types of browsers. |

## The HttpServletRequest Object:

The request object is an instance of a javax.servlet.http.HttpServletRequest object. Each time a client requests a page the JSP engine creates a new object to represent that request.

The request object provides methods to get HTTP header information including form data, cookies, HTTP methods etc.

There are following important methods which can be used to read HTTP header in your JSP program. These method are available with *HttpServletRequest* object which represents client request to webserver.

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **Cookie[] getCookies()** Returns an array containing all of the Cookie objects the client sent with this request. |
| 2 | **Enumeration getAttributeNames()** Returns an Enumeration containing the names of the attributes available to this request. |
| 3 | **Enumeration getHeaderNames()** Returns an enumeration of all the header names this request contains. |
| 4 | **Enumeration getParameterNames()** Returns an Enumeration of String objects containing the names of the parameters contained in this request. |
| 5 | **HttpSession getSession()** Returns the current session associated with this request, or if the request does not have a session, creates one. |
| 6 | **HttpSession getSession(boolean create)** Returns the current HttpSession associated with this request or, if if there is no current session and create is true, returns a new session. |
| 7 | **Locale getLocale()** Returns the preferred Locale that the client will accept content in, based on the Accept-Language header |
| 8 | **Object getAttribute(String name)** Returns the value of the named attribute as an Object, or null if no attribute of the given name exists. |
| 9 | **ServletInputStream getInputStream()** Retrieves the body of the request as binary data using a ServletInputStream. |
| 10 | **String getAuthType()** Returns the name of the authentication scheme used to protect the servlet, for example, "BASIC" or "SSL," or null if the JSP was not protected |
| 11 | **String getCharacterEncoding()** Returns the name of the character encoding used in the body of this request. |
| 12 | **String getContentType()** Returns the MIME type of the body of the request, or null if the type is not known. |
| 13 | **String getContextPath()** Returns the portion of the request URI that indicates the context of the request. |
| 14 | **String getHeader(String name)** Returns the value of the specified request header as a String. |
| 15 | **String getMethod()** Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT. |
| 16 | **String getParameter(String name)** Returns the value of a request parameter as a String, or null if the parameter does not exist. |
| 17 | **String getPathInfo()** Returns any extra path information associated with the URL the client sent when it made this request. |
| 18 | **String getProtocol()** Returns the name and version of the protocol the request. |
| 19 | **String getQueryString()** Returns the query string that is contained in the request URL after the path. |
| 20 | **String getRemoteAddr()** Returns the Internet Protocol (IP) address of the client that sent the request. |
| 21 | **String getRemoteHost()** Returns the fully qualified name of the client that sent the request. |
| 22 | **String getRemoteUser()** Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated. |
| 23 | **String getRequestURI()** Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request. |
| 24 | **String getRequestedSessionId()** Returns the session ID specified by the client. |
| 25 | **String getServletPath()** Returns the part of this request's URL that calls the JSP. |
| 26 | **String[] getParameterValues(String name)** Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist. |
| 27 | **boolean isSecure()** Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS. |
| 28 | **int getContentLength()** Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known. |
| 29 | **int getIntHeader(String name)** Returns the value of the specified request header as an int. |
| 30 | **int getServerPort()** Returns the port number on which this request was received. |

## HTTP Header Request Example:

Following is the example which uses **getHeaderNames()** method of HttpServletRequest to read the HTTP header infromation. This method returns an Enumeration that contains the header information associated with the current HTTP request.

Once we have an Enumeration, we can loop down the Enumeration in the standard manner, using*hasMoreElements()* method to determine when to stop and using *nextElement()* method to get each parameter name.

<%@ page import="java.io.\*,java.util.\*" %>

<html>

<head>

<title>HTTP Header Request Example</title>

</head>

<body>

<center>

<h2>HTTP Header Request Example</h2>

<table width="100%" border="1" align="center">

<tr bgcolor="#949494">

<th>Header Name</th><th>Header Value(s)</th>

</tr>

<%

Enumeration headerNames = request.getHeaderNames();

while(headerNames.hasMoreElements()) {

String paramName = (String)headerNames.nextElement();

out.print("<tr><td>" + paramName + "</td>\n");

String paramValue = request.getHeader(paramName);

out.println("<td> " + paramValue + "</td></tr>\n");

}

%>

</table>

</center>

</body>

</html>

Now put the above code in main.jsp and try to access it. This would produce result something as follows:

# **HTTP Header Request Example**

|  |  |
| --- | --- |
| **Header Name** | **Header Value(s)** |
| accept | \*/\* |
| accept-language | en-us |
| user-agent | Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; InfoPath.2; MS-RTC LM 8) |
| accept-encoding | gzip, deflate |
| host | localhost:8080 |
| connection | Keep-Alive |
| cache-control | no-cache |

To become more comfortable with other methods you can try few more above listed methods in the same fashion.

When a Web server responds to a HTTP request to the browser, the response typically consists of a status line, some response headers, a blank line, and the document. A typical response looks like this:

HTTP/1.1 200 OK

Content-Type: text/html

Header2: ...

...

HeaderN: ...

(Blank Line)

<!doctype ...>

<html>

<head>...</head>

<body>

...

</body>

</html>

The status line consists of the HTTP version (HTTP/1.1 in the example), a status code (200 in the example), and a very short message corresponding to the status code (OK in the example).

Following is a summary of the most useful HTTP 1.1 response headers which go back to the browser from web server side and you would use them very frequently in web programming:

|  |  |
| --- | --- |
| **Header** | **Description** |
| Allow | This header specifies the request methods (GET, POST, etc.) that the server supports. |
| Cache-Control | This header specifies the circumstances in which the response document can safely be cached. It can have values **public, private** or**no-cache** etc. Public means document is cacheable, Private means document is for a single user and can only be stored in private (nonshared) caches and no-cache means document should never be cached. |
| Connection | This header instructs the browser whether to use persistent in HTTP connections or not. A value of **close** instructs the browser not to use persistent HTTP connections and **keep-alive** means using persistent connections. |
| Content-Disposition | This header lets you request that the browser ask the user to save the response to disk in a file of the given name. |
| Content-Encoding | This header specifies the way in which the page was encoded during transmission. |
| Content-Language | This header signifies the language in which the document is written. For example en, en-us, ru, etc. |
| Content-Length | This header indicates the number of bytes in the response. This information is needed only if the browser is using a persistent (keep-alive) HTTP connection. |
| Content-Type | This header gives the MIME (Multipurpose Internet Mail Extension) type of the response document. |
| Expires | This header specifies the time at which the content should be considered out-of-date and thus no longer be cached. |
| Last-Modified | This header indicates when the document was last changed. The client can then cache the document and supply a date by an **If-Modified-Since** request header in later requests. |
| Location | This header should be included with all responses that have a status code in the 300s. This notifies the browser of the document address. The browser automatically reconnects to this location and retrieves the new document. |
| Refresh | This header specifies how soon the browser should ask for an updated page. You can specify time in number of seconds after which a page would be refreshed. |
| Retry-After | This header can be used in conjunction with a 503 (Service Unavailable) response to tell the client how soon it can repeat its request. |
| Set-Cookie | This header specifies a cookie associated with the page. |

## The HttpServletResponse Object:

The response object is an instance of a javax.servlet.http.HttpServletResponse object. Just as the server creates the request object, it also creates an object to represent the response to the client.

The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes etc.

There are following methods which can be used to set HTTP response header in your servlet program. These method are available with *HttpServletResponse* object which represents server response.

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **String encodeRedirectURL(String url)** Encodes the specified URL for use in the sendRedirect method or, if encoding is not needed, returns the URL unchanged. |
| 2 | **String encodeURL(String url)** Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged. |
| 3 | **boolean containsHeader(String name)** Returns a boolean indicating whether the named response header has already been set. |
| 4 | **boolean isCommitted()** Returns a boolean indicating if the response has been committed. |
| 5 | **void addCookie(Cookie cookie)** Adds the specified cookie to the response. |
| 6 | **void addDateHeader(String name, long date)** Adds a response header with the given name and date-value. |
| 7 | **void addHeader(String name, String value)** Adds a response header with the given name and value. |
| 8 | **void addIntHeader(String name, int value)** Adds a response header with the given name and integer value. |
| 9 | **void flushBuffer()** Forces any content in the buffer to be written to the client. |
| 10 | **void reset()** Clears any data that exists in the buffer as well as the status code and headers. |
| 11 | **void resetBuffer()** Clears the content of the underlying buffer in the response without clearing headers or status code. |
| 12 | **void sendError(int sc)** Sends an error response to the client using the specified status code and clearing the buffer. |
| 13 | **void sendError(int sc, String msg)** Sends an error response to the client using the specified status. |
| 14 | **void sendRedirect(String location)** Sends a temporary redirect response to the client using the specified redirect location URL. |
| 15 | **void setBufferSize(int size)** Sets the preferred buffer size for the body of the response. |
| 16 | **void setCharacterEncoding(String charset)** Sets the character encoding (MIME charset) of the response being sent to the client, for example, to UTF-8. |
| 17 | **void setContentLength(int len)** Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header. |
| 18 | **void setContentType(String type)** Sets the content type of the response being sent to the client, if the response has not been committed yet. |
| 19 | **void setDateHeader(String name, long date)** Sets a response header with the given name and date-value. |
| 20 | **void setHeader(String name, String value)** Sets a response header with the given name and value. |
| 21 | **void setIntHeader(String name, int value)** Sets a response header with the given name and integer value. |
| 22 | **void setLocale(Locale loc)** Sets the locale of the response, if the response has not been committed yet. |
| 23 | **void setStatus(int sc)** Sets the status code for this response. |

## HTTP Header Response Example:

Following example would use **setIntHeader()** method to set **Refresh** header to simulate a digital clock:

<%@ page import="java.io.\*,java.util.\*" %>

<html>

<head>

<title>Auto Refresh Header Example</title>

</head>

<body>

<center>

<h2>Auto Refresh Header Example</h2>

<%

// Set refresh, autoload time as 5 seconds

response.setIntHeader("Refresh", 5);

// Get current time

Calendar calendar = new GregorianCalendar();

String am\_pm;

int hour = calendar.get(Calendar.HOUR);

int minute = calendar.get(Calendar.MINUTE);

int second = calendar.get(Calendar.SECOND);

if(calendar.get(Calendar.AM\_PM) == 0)

am\_pm = "AM";

else

am\_pm = "PM";

String CT = hour+":"+ minute +":"+ second +" "+ am\_pm;

out.println("Current Time is: " + CT + "\n");

%>

</center>

</body>

</html>

Now put the above code in main.jsp and try to access it. This would display current system time after every 5 seconds as follows. Just run the JSP and wait to see the result:

|  |
| --- |
| Auto Refresh Header Example Current Time is: 9:44:50 PM |

To become more comfortable with other methods you can try few more above listed methods in the same fashion.

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your backend program. The browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

## GET method:

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows:

http://www.test.com/hello?key1=value1&key2=value2

The GET method is the default method to pass information from browser to web server and it produces a long string that appears in your browser's Location:box. Never use the GET method if you have password or other sensitive information to pass to the server.

The GET method has size limitation: only 1024 characters can be in a request string.

This information is passed using QUERY\_STRING header and will be accessible through QUERY\_STRING environment variable which can be handled using getQueryString() and getParameter() methods of request object.

## POST method:

A generally more reliable method of passing information to a backend program is the POST method.

This method packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL it sends it as a separate message. This message comes to the backend program in the form of the standard input which you can parse and use for your processing.

JSP handles this type of requests using getParameter() method to read simple parameters and getInputStream() method to read binary data stream coming from the client.

## Reading Form Data using JSP

JSP handles form data parsing automatically using the following methods depending on the situation:

* **getParameter():** You call request.getParameter() method to get the value of a form parameter.
* **getParameterValues():** Call this method if the parameter appears more than once and returns multiple values, for example checkbox.
* **getParameterNames():** Call this method if you want a complete list of all parameters in the current request.
* **getInputStream():** Call this method to read binary data stream coming from the client.

## GET Method Example Using URL:

Here is a simple URL which will pass two values to HelloForm program using GET method.

**http://localhost:8080/main.jsp?first\_name=ZARA&last\_name=ALI**

Below is **main.jsp** JSP program to handle input given by web browser. We are going to use**getParameter()** method which makes it very easy to access passed information:

<html>

<head>

<title>Using GET Method to Read Form Data</title>

</head>

<body>

<center>

<h1>Using GET Method to Read Form Data</h1>

<ul>

<li><p><b>First Name:</b>

<%= request.getParameter("first\_name")%>

</p></li>

<li><p><b>Last Name:</b>

<%= request.getParameter("last\_name")%>

</p></li>

</ul>

</body>

</html>

Now type *http://localhost:8080/main.jsp?first\_name=ZARA&last\_name=ALI* in your browser's Location:box. This would generate following result:

|  |
| --- |
| **Using GET Method to Read Form Data**  * **First Name**: ZARA * **Last Name**: ALI |

## GET Method Example Using Form:

Here is a simple example which passes two values using HTML FORM and submit button. We are going to use same JSP main.jsp to handle this input.

<html>

<body>

<form action="main.jsp" method="GET">

First Name: <input type="text" name="first\_name">

<br />

Last Name: <input type="text" name="last\_name" />

<input type="submit" value="Submit" />

</form>

</body>

</html>

Keep this HTML in a file Hello.htm and put it in <Tomcat-installation-directory>/webapps/ROOT directory. When you would access *http://localhost:8080/Hello.htm*, here is the actual output of the above form.

Top of Form

First Name:    
Last Name: 

Bottom of Form

Try to enter First Name and Last Name and then click submit button to see the result on your local machine where tomcat is running. Based on the input provided, it will generate similar result as mentioned in the above example.

## POST Method Example Using Form:

Let us do little modification in the above JSP to handle GET as well as POST methods. Below is **main.jsp**JSP program to handle input given by web browser using GET or POST methods.

Infact there is no change in above JSP because only way of passing parameters is changed and no binary data is being passed to the JSP program. File handling related concepts would be explained in separate chapter where we need to read binary data stream.

<html>

<head>

<title>Using GET and POST Method to Read Form Data</title>

</head>

<body>

<center>

<h1>Using GET Method to Read Form Data</h1>

<ul>

<li><p><b>First Name:</b>

<%= request.getParameter("first\_name")%>

</p></li>

<li><p><b>Last Name:</b>

<%= request.getParameter("last\_name")%>

</p></li>

</ul>

</body>

</html>

Following is the content of Hello.htm file:

<html>

<body>

<form action="main.jsp" method="POST">

First Name: <input type="text" name="first\_name">

<br />

Last Name: <input type="text" name="last\_name" />

<input type="submit" value="Submit" />

</form>

</body>

</html>

Now let us keep main.jsp and hello.htm in <Tomcat-installation-directory>/webapps/ROOT directory. When you would access *http://localhost:8080/Hello.htm*, below is the actual output of the above form.

Top of Form

First Name:    
Last Name: 

Bottom of Form

Try to enter First and Last Name and then click submit button to see the result on your local machine where tomcat is running.

Based on the input provided, it would generate similar result as mentioned in the above examples.

## Passing Checkbox Data to JSP Program

Checkboxes are used when more than one option is required to be selected.

Here is example HTML code, CheckBox.htm, for a form with two checkboxes

<html>

<body>

<form action="main.jsp" method="POST" target="\_blank">

<input type="checkbox" name="maths" checked="checked" /> Maths

<input type="checkbox" name="physics" /> Physics

<input type="checkbox" name="chemistry" checked="checked" />

Chemistry

<input type="submit" value="Select Subject" />

</form>

</body>

</html>

The result of this code is the following form

Top of Form

 Maths  Physics  Chemistry

Bottom of Form

Below is main.jsp JSP program to handle input given by web browser for checkbox button.

<html>

<head>

<title>Reading Checkbox Data</title>

</head>

<body>

<center>

<h1>Reading Checkbox Data</h1>

<ul>

<li><p><b>Maths Flag:</b>

<%= request.getParameter("maths")%>

</p></li>

<li><p><b>Physics Flag:</b>

<%= request.getParameter("physics")%>

</p></li>

<li><p><b>Chemistry Flag:</b>

<%= request.getParameter("chemistry")%>

</p></li>

</ul>

</body>

</html>

For the above example, it would display following result:

|  |
| --- |
| **Reading Checkbox Data**  * **Maths Flag :**: on * **Physics Flag:**: null * **Chemistry Flag:**: on |

## Reading All Form Parameters:

Following is the generic example which uses **getParameterNames()** method of HttpServletRequest to read all the available form parameters. This method returns an Enumeration that contains the parameter names in an unspecified order.

Once we have an Enumeration, we can loop down the Enumeration in the standard manner, using*hasMoreElements()* method to determine when to stop and using *nextElement()* method to get each parameter name.

<%@ page import="java.io.\*,java.util.\*" %>

<html>

<head>

<title>HTTP Header Request Example</title>

</head>

<body>

<center>

<h2>HTTP Header Request Example</h2>

<table width="100%" border="1" align="center">

<tr bgcolor="#949494">

<th>Param Name</th><th>Param Value(s)</th>

</tr>

<%

Enumeration paramNames = request.getParameterNames();

while(paramNames.hasMoreElements()) {

String paramName = (String)paramNames.nextElement();

out.print("<tr><td>" + paramName + "</td>\n");

String paramValue = request.getHeader(paramName);

out.println("<td> " + paramValue + "</td></tr>\n");

}

%>

</table>

</center>

</body>

</html>

Following is the content of Hello.htm:

<html>

<body>

<form action="main.jsp" method="POST" target="\_blank">

<input type="checkbox" name="maths" checked="checked" /> Maths

<input type="checkbox" name="physics" /> Physics

<input type="checkbox" name="chemistry" checked="checked" /> Chem

<input type="submit" value="Select Subject" />

</form>

</body>

</html>

Now try calling JSP using above Hello.htm, this would generate a result something like as below based on the provided input:

# **Reading All Form Parameters**

|  |  |
| --- | --- |
| **Param Name** | **Param Value(s)** |
| maths | on |
| chemistry | on |

You can try above JSP to read any other form's data which is having other objects like text box, radio button or drop down box etc.