tutorialspoint
SIMPLYEASYLEARNING

JSP - Quick Guide

JSP - Overview

What is JavaServer Pages?

JavaServer Pages (JSP) is a technology for developing Webpages that supports dynamic content. This helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>.

A JavaServer Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.

Using JSP, you can collect input from users through Webpage forms, present records from a database or another source, and create Webpages dynamically.

JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages, and sharing information between requests, pages etc.

Why Use JSP?

JavaServer Pages often serve the same purpose as programs implemented using the **Common Gateway Interface (CGI)**. But JSP offers several advantages in comparison with the CGI.

- Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself instead of having separate CGI files.
- JSP are always compiled before they are processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested.

JavaServer Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including **JDBC, JNDI, EJB, JAXP,** etc.

JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

Finally, JSP is an integral part of Java EE, a complete platform for enterprise class applications. This means that JSP can play a part in the simplest applications to the most complex and demanding.

Advantages of JSP

Following table lists out the other advantages of using JSP over other technologies −

vs. Active Server Pages (ASP)

The advantages of JSP are twofold. First, the dynamic part is written in Java, not Visual Basic or other MS specific language, so it is more powerful and easier to use. Second, it is portable to other operating systems and non-Microsoft Web servers.

vs. Pure Servlets

It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.

vs. Server-Side Includes (SSI)

SSI is really only intended for simple inclusions, not for "real" programs that use form data, make database connections, and the like.

vs. JavaScript

JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.

vs. Static HTML

Regular HTML, of course, cannot contain dynamic information.

What is Next?

I would take you step by step to set up your environment to start with JSP. I'm assuming you have good hands-on with Java Programming to proceed with learning JSP.

If you are not aware of Java Programming Language, then we would recommend you go through our Java Tutorial to understand Java Programming.

JSP - Environment Setup

A development environment is where you would develop your JSP programs, test them and finally run them.

This tutorial will guide you to setup your JSP development environment which involves the following steps −

Setting up Java Development Kit

This step involves downloading an implementation of the Java Software Development Kit (SDK) and setting up the PATH environment variable appropriately.

You can download SDK from Oracle's Java site − Java SE Downloads .

Once you download your Java implementation, follow the given instructions to install and configure the setup. Finally set the **PATH and JAVA_HOME** environment variables to refer to the directory that contains **java** and **javac**, typically **java_install_dir/bin** and **java_install_dir** respectively.

If you are running Windows and install the SDK in **C:\jdk1.5.0_20**, you need to add the following line in your **C:\autoexec.bat** file.

```
set PATH = C:\jdk1.5.0_20\bin;%PATH%
set JAVA_HOME = C:\jdk1.5.0_20
```

Alternatively, on **Windows NT/2000/XP**, you can also right-click on **My Computer**, select **Properties**, then **Advanced**, followed by **Environment Variables**. Then, you would update the PATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if the SDK is installed in **/usr/local/jdk1.5.0_20** and you use the C shell, you will put the following into your **.cshrc** file.

```
setenv PATH /usr/local/jdk1.5.0_20/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.5.0_20
```

Alternatively, if you use an **Integrated Development Environment (IDE)** like **Borland JBuilder, Eclipse, IntelliJ IDEA**, or **Sun ONE Studio**, compile and run a simple program to confirm that the IDE knows where you installed Java.

Setting up Web Server: Tomcat

A number of Web Servers that support JavaServer Pages and Servlets development are available in the market. Some web servers can be downloaded for free and Tomcat is one of them.

Apache Tomcat is an open source software implementation of the JavaServer Pages and Servlet technologies and can act as a standalone server for testing JSP and Servlets, and can be integrated with the Apache Web Server. Here are the steps to set up Tomcat on your machine −

Download the latest version of Tomcat from https://tomcat.apache.org/ .

Once you downloaded the installation, unpack the binary distribution into a convenient location. For example, in **C:\apache-tomcat-5.5.29 on windows, or /usr/local/apache-tomcat-5.5.29** on Linux/Unix and create **CATALINA_HOME** environment variable pointing to these locations.

Tomcat can be started by executing the following commands on the Windows machine −

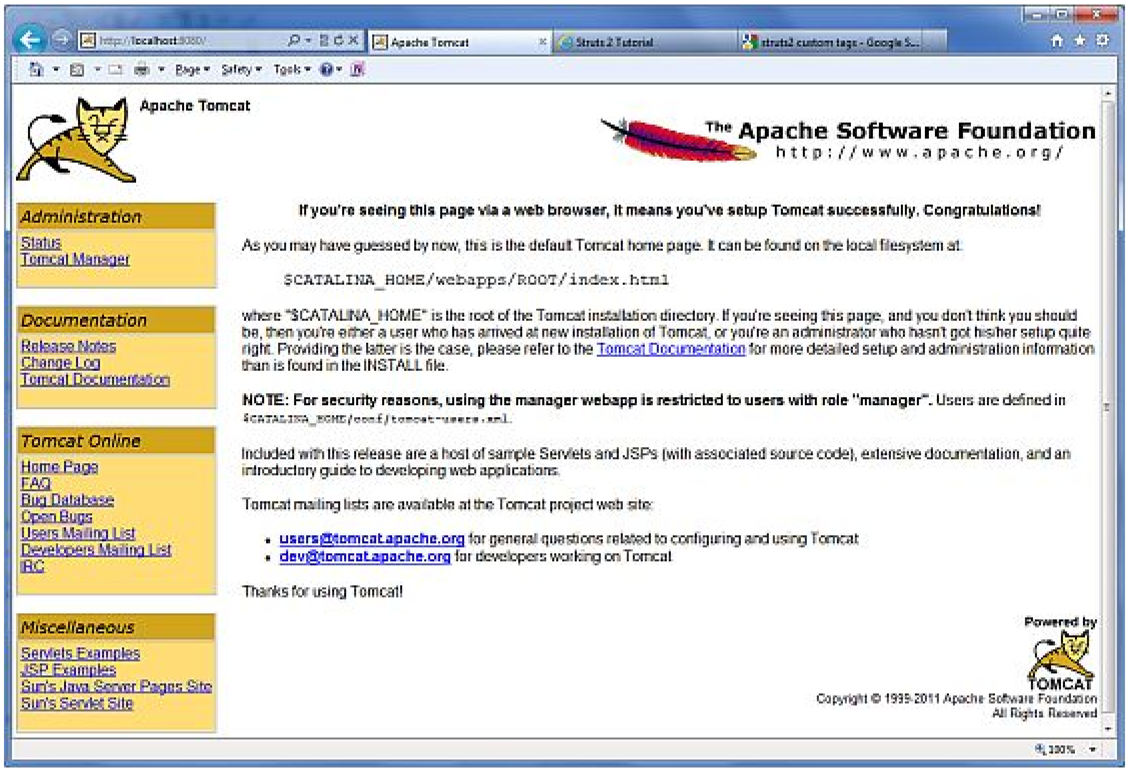
```
%CATALINA_HOME%\bin\startup.bat

or

C:\apache-tomcat-5.5.29\bin\startup.bat
```

Tomcat can be started by executing the following commands on the Unix (Solaris, Linux, etc.) machine −

```
$CATALINA_HOME/bin/startup.sh

or

/usr/local/apache-tomcat-5.5.29/bin/startup.sh
```

After a successful startup, the default web-applications included with Tomcat will be available by visiting **http://localhost:8080/**.

Upon execution, you will receive the following output −



Apache Tomcat

The Apache Software Foundation
http://www.apache.org/

Administration
Status
Tomcat Manager

Documentation
Release Notes
Change Log
Tomcat Documentation

Tomcat Online
Home Page
FAQ
Bug Database
Open Bugs
Users Mailing List
Developers Mailing List
IRC

Miscellaneous
Servlets Examples
JSP Examples
Sun's Java Server Pages Site
Sun's Servlet Site

If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!

As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:

```
$CATALINA_HOME/webapps/ROOT/index.html
```

where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and administration information than is found in the INSTALL file.

NOTE: For security reasons, using the manager webapp is restricted to users with role "manager". Users are defined in `$CATALINA_HOME/conf/tomcat-users.xml`.

Included with this release are a host of sample Servlets and JSPs (with associated source code), extensive documentation, and an introductory guide to developing web applications.

Tomcat mailing lists are available at the Tomcat project web site:

- users@tomcat.apache.org for general questions related to configuring and using Tomcat
- dev@tomcat.apache.org for developers working on Tomcat

Thanks for using Tomcat!

Powered by
TOMCAT
Copyright © 1999-2011 Apache Software Foundation
All Rights Reserved

Further information about configuring and running Tomcat can be found in the documentation included here, as well as on the Tomcat web site −

https://tomcat.apache.org/ .

Tomcat can be stopped by executing the following commands on the Windows machine –

```
%CATALINA_HOME%\bin\shutdown
or

C:\apache-tomcat-5.5.29\bin\shutdown
```

Tomcat can be stopped by executing the following commands on Unix (Solaris, Linux, etc.) machine –

```
$CATALINA_HOME/bin/shutdown.sh

or

/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh
```

Setting up CLASSPATH

Since servlets are not part of the Java Platform, Standard Edition, you must identify the servlet classes to the compiler.

If you are running Windows, you need to put the following lines in your **C:\autoexec.bat** file.

```
set CATALINA = C:\apache-tomcat-5.5.29
set CLASSPATH = %CATALINA%\common\lib\jsp-api.jar;%CLASSPATH%
```

Alternatively, on **Windows NT/2000/XP**, you can also right-click on **My Computer**, select **Properties**, then **Advanced**, then **Environment Variables**. Then, you would update the CLASSPATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if you are using the C shell, you would put the following lines into your **.cshrc** file.

```
setenv CATALINA = /usr/local/apache-tomcat-5.5.29
setenv CLASSPATH $CATALINA/common/lib/jsp-api.jar:$CLASSPATH
```

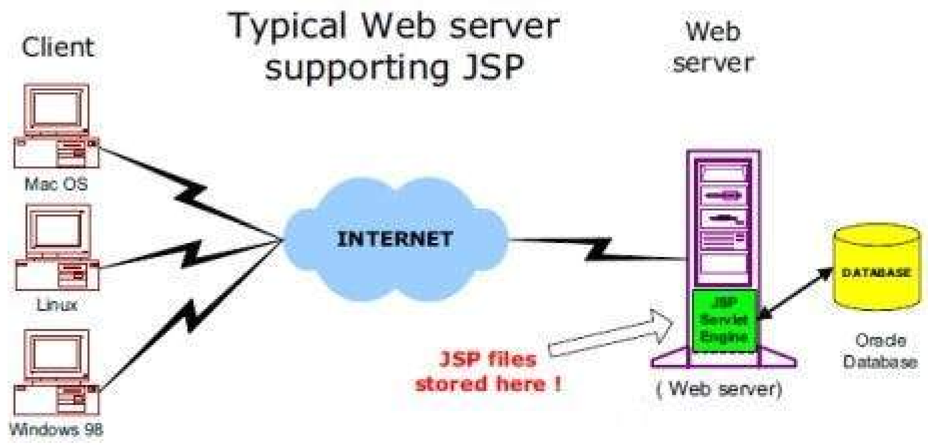
NOTE – Assuming that your development directory is **C:\JSPDev (Windows)** or **/usr/JSPDev (Unix)**, then you would need to add these directories as well in CLASSPATH.

JSP - Architecture

The web server needs a JSP engine, i.e, a container to process JSP pages. The JSP container is responsible for intercepting requests for JSP pages. This tutorial makes use of Apache which has built-in JSP container to support JSP pages development.

A JSP container works with the Web server to provide the runtime environment and other services a JSP needs. It knows how to understand the special elements that are part of JSPs.

Following diagram shows the position of JSP container and JSP files in a Web application.



JSP Processing

The following steps explain how the web server creates the Webpage using JSP −

As with a normal page, your browser sends an HTTP request to the web server.

The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with **.jsp** instead of **.html**.

The JSP engine loads the JSP page from disk and converts it into a servlet content. This conversion is very simple in which all template text is converted to println( ) statements and all JSP elements are converted to Java code. This code implements the corresponding dynamic behavior of the page.

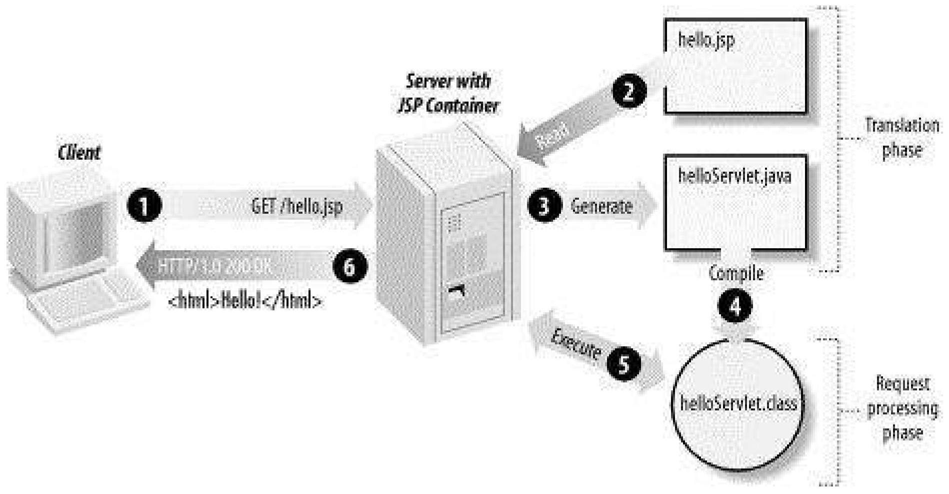
The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.

A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format. The output is furthur passed on to the web server by the servlet engine inside an HTTP response.

The web server forwards the HTTP response to your browser in terms of static HTML content.

Finally, the web browser handles the dynamically-generated HTML page inside the HTTP response exactly as if it were a static page.

All the above mentioned steps can be seen in the following diagram −



Typically, the JSP engine checks to see whether a servlet for a JSP file already exists and whether the modification date on the JSP is older than the servlet. If the JSP is older than its generated servlet, the JSP container assumes that the JSP hasn't changed and that the generated servlet still matches the JSP's contents. This makes the process more efficient than with the other scripting languages (such as PHP) and therefore faster.

So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming wiz. Except for the translation phase, a JSP page is handled exactly like a regular servlet.

JSP - Lifecycle

In this chapter, we will discuss the lifecycle of JSP. The key to understanding the low-level functionality of JSP is to understand the simple life cycle they follow.

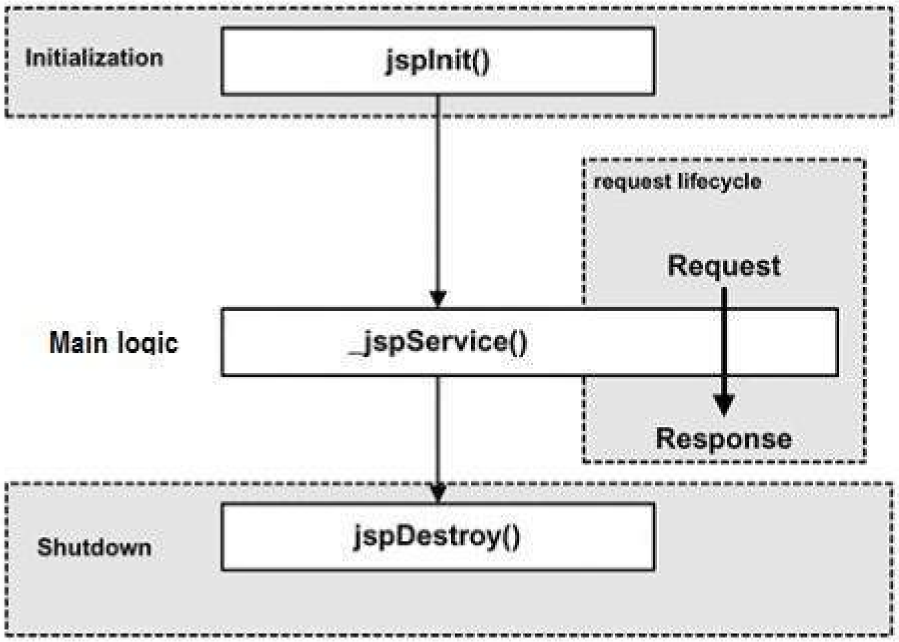
A JSP life cycle is defined as the process from its creation till the destruction. This is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet.

Paths Followed By JSP

The following are the paths followed by a JSP −

- Compilation
- Initialization
- Execution
- Cleanup

The four major phases of a JSP life cycle are very similar to the Servlet Life Cycle. The four phases have been described below −



JSP Compilation

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

The compilation process involves three steps −

- Parsing the JSP.
- Turning the JSP into a servlet.
- Compiling the servlet.

JSP Initialization

When a container loads a JSP it invokes the **jspInit()** method before servicing any requests. If you need to perform JSP-specific initialization, override the **jspInit()** method −

```
public void jspInit(){
   // Initialization code...
}
```

Typically, initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

JSP Execution

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the **_jspService()** method in the JSP.

The _jspService() method takes an **HttpServletRequest** and an **HttpServletResponse** as its parameters as follows −

```
void _jspService(HttpServletRequest request, HttpServletResponse response) {
   // Service handling code...
}
```

The **_jspService()** method of a JSP is invoked on request basis. This is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods, i.e, **GET, POST, DELETE**, etc.

JSP Cleanup

The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.

The **jspDestroy()** method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.

The jspDestroy() method has the following form −

```
public void jspDestroy() {
   // Your cleanup code goes here.
}
```

JSP - Syntax

In this chapter, we will discuss Syntax in JSP. We will understand the basic use of simple syntax (i.e, elements) involved with JSP development.

Elements of JSP

The elements of JSP have been described below −

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 the XML equivalent of the above syntax as follows –

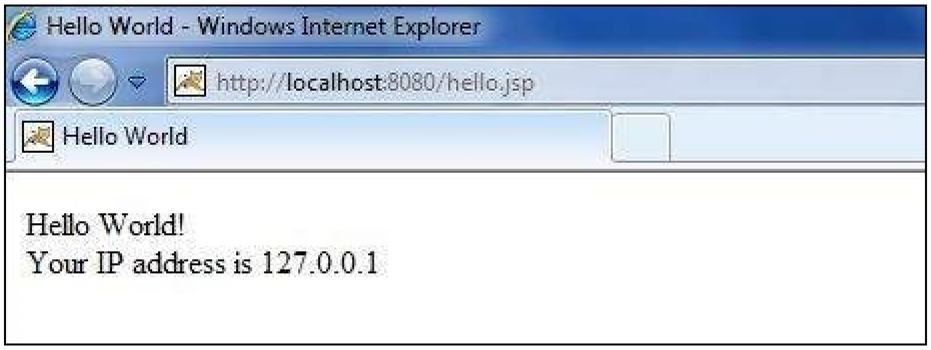
```
<jsp:scriptlet>
   code fragment
</jsp:scriptlet>
```

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 the above code in JSP file **hello.jsp** and put this file in **C:\apache-tomcat7.0.2\webapps\ROOT** directory. Browse through the same using URL **http://localhost:8080/hello.jsp**. The above code will generate the following result –


Hello World - Windows Internet Explorer
http://localhost:8080/hello.jsp
Hello World

```
Hello World!
Your IP address is 127.0.0.1
```

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 for JSP Declarations –

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

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

```
<jsp:declaration>
   code fragment
</jsp:declaration>
```

Following is an 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 the XML equivalent of the above syntax as follows −

```
<jsp:expression>
   expression
</jsp:expression>
```

Following example shows a JSP Expression −

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

   <body>
      <p>Today's date: <%= (new java.util.Date()).toLocaleString()%></p>
   </body>
</html>
```

The above code will generate the 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", a part of your JSP page.

Following is the syntax of the JSP comments −

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

Following example shows the 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>
```

The above code will generate the 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 −

S.No.	Syntax & Purpose
1	**<%-- comment --%>** A JSP comment. Ignored by the JSP engine.
2	**<!-- comment -->** An HTML comment. Ignored by the browser.
3	**<\%** Represents static <% literal.
4	**%\>** Represents static %> literal.
5	**\'** A single quote in an attribute that uses single quotes.
6	**\"** 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 −

S.No.	Directive & Description
1	**<%@ page ... %>** Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
2	**<%@ include ... %>** Includes a file during the translation phase.
3	**<%@ taglib ... %>** Declares a tag library, containing custom actions, used in the page

We would explain the JSP directive in a separate chapter JSP - Directives

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. Following table lists out the available JSP Actions −

S.No.	Syntax & Purpose
1	**jsp:include** Includes a file at the time the page is requested.
2	**jsp:useBean** Finds or instantiates a JavaBean.
3	**jsp:setProperty** Sets the property of a JavaBean.

4	**jsp:getProperty** Inserts the property of a JavaBean into the output.
5	**jsp:forward** Forwards the requester to a new page.
6	**jsp:plugin** Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin.
7	**jsp:element** Defines XML elements dynamically.
8	**jsp:attribute** Defines dynamically-defined XML element's attribute.
9	**jsp:body** Defines dynamically-defined XML element's body.
10	**jsp:text** Used to write template text in JSP pages and documents.

We would explain JSP actions in a separate chapter JSP - Actions

JSP Implicit Objects

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

S.No.	Object & Description
1	**request** This is the **HttpServletRequest** object associated with the request.
2	**response** This is the **HttpServletResponse** object associated with the response to the client.

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

We would explain JSP Implicit Objects in a separate chapter JSP - Implicit Objects .

Control-Flow Statements

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 the 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>
```

The above code will generate the 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>
```

The above code will generate the 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>
```

The above code will generate the following result −

```
JSP Tutorial

JSP Tutorial

JSP Tutorial
```

Above example can be written using the **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>
```

The above code will generate the following result −

```
JSP Tutorial
JSP Tutorial


JSP Tutorial
```

JSP Operators

JSP supports all the logical and arithmetic operators supported by Java. Following table lists out 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

In this chapter, we will discuss Directives in JSP. These directives provide directions and instructions to the container, telling it how to handle certain aspects of the 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 −

S.No.	Directive & Description
1	**<%@ page ... %>** Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
2	**<%@ include ... %>** Includes a file during the translation phase.
3	**<%@ taglib ... %>** Declares a tag library, containing custom actions, used in the page

JSP - The page Directive

The **page** directive is used to provide instructions to the container. These instructions 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 the page directive −

```
<%@ page attribute = "value" %>
```

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

```
<jsp:directive.page attribute = "value" />
```

Attributes

Following table lists out the attributes associated with the page directive −

S.No.	Attribute & Purpose
1	**buffer** Specifies a buffering model for the output stream.
2	**autoFlush** Controls the behavior of the servlet output buffer.
3	**contentType** Defines the character encoding scheme.
4	**errorPage** Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
5	**isErrorPage** Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.
6	**extends** Specifies a superclass that the generated servlet must extend.
7	**import** Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.

8	**info** Defines a string that can be accessed with the servlet's **getServletInfo()** method.
9	**isThreadSafe** Defines the threading model for the generated servlet.
10	**language** Defines the programming language used in the JSP page.
11	**session** Specifies whether or not the JSP page participates in HTTP sessions
12	**isELIgnored** Specifies whether or not the EL expression within the JSP page will be ignored.
13	**isScriptingEnabled** Determines if the scripting elements are allowed for use.

Check for more details related to all the above attributes at Page Directive .

The include Directive

The **include** directive is used to include 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 the ***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 the XML equivalent of the above syntax as follows −

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

For more details related to include directive, check the Include Directive .

The taglib Directive

The JavaServer Pages API allow 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 means for identifying the custom tags in your JSP page.

The taglib directive follows the syntax given below –

```
<%@ taglib uri="uri" prefix = "prefixOfTag" >
```

Here, 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 the XML equivalent of the above syntax as follows –

```
<jsp:directive.taglib uri = "uri" prefix = "prefixOfTag" />
```

For more details related to the taglib directive, check the Taglib Directive .

JSP - Actions

In this chapter, we will discuss Actions in JSP. These 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. The following table lists out the available JSP actions –

S.No.	Syntax & Purpose
1	**jsp:include** Includes a file at the time the page is requested.
2	**jsp:useBean** Finds or instantiates a JavaBean.
3	**jsp:setProperty** Sets the property of a JavaBean.

4	**jsp:getProperty** Inserts the property of a JavaBean into the output.
5	**jsp:forward** Forwards the requester to a new page.
6	**jsp:plugin** Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin.
7	**jsp:element** Defines XML elements dynamically.
8	**jsp:attribute** Defines dynamically-defined XML element's attribute.
9	**jsp:body** Defines dynamically-defined XML element's body.
10	**jsp:text** Used 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 table lists out the attributes associated with the include action −

S.No.	Attribute & Description
1	**page** The relative URL of the page to be included.
2	**flush** The boolean attribute determines whether the included resource has its buffer flushed before it is included.

Example

Let us define the following two files **(a)date.jsp** and **(b) main.jsp** as follows −

Following is the content of the **date.jsp** file −

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

Following is the content of the **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>
```

Let us now keep all these files in the root directory and try to access **main.jsp**. You will receive the following output −

```
                 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 the bean properties.

Following table lists out the attributes associated with the useBean action −

S.No.	Attribute & Description
1	**class** Designates the full package name of the bean.
2	**type** Specifies the type of the variable that will refer to the object.
3	**beanName** Gives the name of the bean as specified by the instantiate () method of the java.beans.Beans class.

Let us now discuss the **jsp:setProperty** and the **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 given 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 given 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 table lists out the attributes associated with the **setProperty** action −

S.No.	Attribute & Description
1	**name** Designates the bean the property of which will be set. The Bean must have been previously defined.
2	**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.
3	**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.
4	**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. The syntax of the getProperty action is as follows −

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

Following table lists out the required attributes associated with the **getProperty** action −

S.No.	Attribute & Description
1	**name** The name of the Bean that has a property to be retrieved. The Bean must have been previously defined.
2	**property** The property attribute is the name of the Bean property to be retrieved.

Example

Let us define a test bean that will further be used 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 the above code to the generated **TestBean.class** file and make sure that you copied the TestBean.class in **C:\apache-tomcat-7.0.2\webapps\WEB-INF\classes\action** folder and the **CLASSPATH** variable should also be set to this folder −

Now use the following code in **main.jsp** file. This 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>
```

Let us now try to access **main.jsp**, it would display the 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.

Following is the syntax of the **forward** action −

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

Following table lists out the required attributes associated with the forward action −

S.No.	Attribute & Description
1	**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 the following two files **(a) date.jsp** and **(b) main.jsp** as follows −

Following is the content of the **date.jsp** file −

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

Following is the content of the **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>
```

Let us now keep all these files in the root directory and try to access **main.jsp**. This would display result something like as below.

Here it discarded the content from the main page and displayed the 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 the 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>, <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 the 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 the template text in JSP pages and documents. Following is the simple syntax for this action −

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

The body of the template cannot contain other elements; it can only contain text and EL expressions (Note − EL expressions are explained in a 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 the above example with and without **<jsp:text>** action.

JSP - Implicit Objects

In this chapter, we will discuss the Implicit Objects in JSP. These Objects are the Java objects that the JSP Container makes available to the developers in each page and the developer can call them directly without being explicitly declared. JSP Implicit Objects are also called **pre-defined variables**.

Following table lists out the nine Implicit Objects that JSP supports −

S.No.	Object & Description
1	**request** This is the **HttpServletRequest** object associated with the request.
2	**response**

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

The request 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 the HTTP header information including form data, cookies, HTTP methods etc.

We can cover a complete set of methods associated with the request object in a subsequent chapter − JSP - Client Request .

The response 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.

We will cover a complete set of methods associated with the response object in a subsequent chapter − JSP - Server Response .

The out Object

The out implicit object is an instance of a **javax.servlet.jsp.JspWriter** object and is used to send content in a response.

The initial JspWriter object is instantiated differently depending on whether the page is buffered or not. Buffering can be easily turned off by using the **buffered = 'false'** attribute of the page directive.

The JspWriter object contains most of the same methods as the **java.io.PrintWriter** class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws **IOExceptions**.

Following table lists out the important methods that we will use to write **boolean char, int, double, object, String**, etc.

S.No.	Method & Description
1	**out.print(dataType dt)** Print a data type value
2	**out.println(dataType dt)** Print a data type value then terminate the line with new line character.
3	**out.flush()** Flush the stream.

The session Object

The session object is an instance of **javax.servlet.http.HttpSession** and behaves exactly the same way that session objects behave under Java Servlets.

The session object is used to track client session between client requests. We will cover the complete usage of session object in a subsequent chapter − JSP - Session Tracking .

The application Object

The application object is direct wrapper around the **ServletContext** object for the generated Servlet and in reality an instance of a **javax.servlet.ServletContext** object.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the **jspDestroy()** method.

By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it.

We will check the use of Application Object in JSP - Hits Counter chapter.

The config Object

The config object is an instantiation of **javax.servlet.ServletConfig** and is a direct wrapper around the **ServletConfig** object for the generated servlet.

This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc.

The following **config** method is the only one you might ever use, and its usage is trivial −

```
config.getServletName();
```

This returns the servlet name, which is the string contained in the **<servlet-name>** element defined in the **WEB-INF\web.xml** file.

The pageContext Object

The pageContext object is an instance of a **javax.servlet.jsp.PageContext** object. The pageContext object is used to represent the entire JSP page.

This object is intended as a means to access information about the page while avoiding most of the implementation details.

This object stores references to the request and response objects for each request. The **application, config, session**, and out objects are derived by accessing attributes of this object.

The pageContext object also contains information about the directives issued to the JSP page, including the buffering information, the errorPageURL, and page scope.

The PageContext class defines several fields, including **PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE,** and **APPLICATION_SCOPE**, which identify the four scopes. It also supports more than 40 methods, about half of which are inherited from the **javax.servlet.jsp.JspContext class**.

One of the important methods is **removeAttribute**. This method accepts either one or two arguments. For example, **pageContext.removeAttribute ("attrName")** removes the attribute from all scopes, while the following code only removes it from the page scope −

```
pageContext.removeAttribute("attrName", PAGE_SCOPE);
```

The use of pageContext can be checked in JSP - File Uploading chapter.

The page Object

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page.

The page object is really a direct synonym for the **this** object.

The exception Object

The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

We will cover the complete usage of this object in JSP - Exception Handling chapter.

JSP - Client Request

In this chapter, we will discuss Client Request in JSP. When a browser requests for a Webpage, it sends a lot of information to the web server. This information cannot be read directly because this information travels as a part of header of HTTP request. You can check HTTP Protocol for more information on this.

Following table lists out the important header information which comes from the browser. This information is frequently used in web programming −

S.No.	Header & Description
1	**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.
2	**Accept-Charset**

	This header specifies the character sets that the browser can use to display the information. For example, **ISO-8859-1**.
3	**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.
4	**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.
5	**Authorization** This header is used by clients to identify themselves when accessing password-protected webpages.
6	**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.
7	**Content-Length** This header is applicable only to **POST** requests and gives the size of the POST data in bytes.
8	**Cookie** This header returns cookies to servers that previously sent them to the browser.
9	**Host** This header specifies the host and port as given in the original URL.
10	**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.

11	**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.
12	**Referer** This header indicates the URL of the referring webpages. For example, if you are at Webpage 1 and click on a link to Webpage 2, the URL of Webpage 1 is included in the Referer header when the browser requests Webpage 2.
13	**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.

Following table lists out the important methods that can be used to read HTTP header in your JSP program. These methods are available with *HttpServletRequest* object which represents client request to webserver.

S.No.	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 the this request, or if the request does not have a session, creates one.
6	**HttpSession getSession(boolean create)** Returns the current HttpSession associated with the 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 uses.
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 information. 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. We will use the ***hasMoreElements()*** method to determine when to stop and the ***nextElement()*** method to get the name of 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>
```

Let us now put the above code in **main.jsp** and try to access it.

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

You can try working on all the methods in a similar way.

JSP - Server Response

In this chapter, we will discuss the Server Response in JSP. When a Web server responds to a HTTP request, 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 the web server. These headers are frequently used in web programming −

S.No.	Header & Description
1	**Allow** This header specifies the request methods (**GET, POST**, etc.) that the server supports.
2	**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.
3	**Connection** This header instructs the browser whether to use persistent HTTP connections or not. A value of **close** instructs the browser not to use persistent HTTP connections and **keep-alive** means using persistent connections.

4	**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.
5	**Content-Encoding** This header specifies the way in which the page was encoded during transmission.
6	**Content-Language** This header signifies the language in which the document is written. For example, **en, en-us, ru,** etc.
7	**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.
8	**Content-Type** This header gives the **MIME** (**Multipurpose Internet Mail Extension**) type of the response document.
9	**Expires** This header specifies the time at which the content should be considered out-of-date and thus no longer be cached.
10	**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.
11	**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.
12	**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.
13	**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.
14	**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.

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

S.No.	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 also 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 will display the current system time after every 5 seconds as follows. Run the JSP. You will receive the following output: –

```
Auto Refresh Header Example

Current Time is: 9:44:50 PM
```

You can try working out on the other methods in a similar way.

JSP - Http Status Codes

In this chapter, we will discuss the Http Status Codes in JSP. The format of the HTTP request and the HTTP response messages are similar and will have the following structure −

- An initial status line + CRLF (Carriage Return + Line Feed ie. New Line)
- Zero or more header lines + CRLF
- A blank line ie. a CRLF
- An optional message body like file, query data or query output.

For example, a server response header looks like the following −

```
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 table lists out the HTTP status codes and associated messages that might be returned from the Web Server −

Code	Message	Description
100	Continue	Only a part of the request has been received by the server, but as long as it has not been rejected, the client should continue with the request
101	Switching Protocols	The server switches protocol.
200	OK	The request is OK
201	Created	The request is complete, and a new resource is created
202	Accepted	The request is accepted for processing, but the processing is not complete.
203	Non-authoritative	

	Information	
204	No Content	
205	Reset Content	
206	Partial Content	
300	Multiple Choices	A link list; the user can select a link and go to that location. Maximum five addresses.
301	Moved Permanently	The requested page has moved to a new url.
302	Found	The requested page has moved temporarily to a new url.
303	See Other	The requested page can be found under a different url.
304	Not Modified	
305	Use Proxy	
306	*Unused*	This code was used in a previous version. It is no longer used, but the code is reserved.
307	Temporary Redirect	The requested page has moved temporarily to a new url.
400	Bad Request	The server did not understand the request.
401	Unauthorized	The requested page needs a username and a password.
402	Payment Required	*You can not use this code yet.*
403	Forbidden	Access is forbidden to the requested page
404	Not Found	The server can not find the requested page.
405	Method Not Allowed	The method specified in the request is not allowed.
406	Not Acceptable	The server can only generate a response that is not accepted by the client.
407	Proxy Authentication Required	You must authenticate with a proxy server before this request can be served.
408	Request Timeout	The request took longer than the server was prepared to wait.
409	Conflict	The request could not be completed because of a conflict.
410	Gone	The requested page is no longer available.
411	Length Required	The "Content-Length" is not defined. The server will not

		accept the request without it.
412	Precondition Failed	The precondition given in the request evaluated to false by the server.
413	Request Entity Too Large	The server will not accept the request, because the request entity is too large.
414	Request-url Too Long	The server will not accept the request, because the url is too long. This occurs when you convert a "post" request to a "get" request with a long query information.
415	Unsupported Media Type	The server will not accept the request, because the media type is not supported.
417	Expectation Failed	
500	Internal Server Error	The request was not completed. The server met an unexpected condition.
501	Not Implemented	The request was not completed. The server did not support the functionality required.
502	Bad Gateway	The request was not completed. The server received an invalid response from the upstream server.
503	Service Unavailable	The request was not completed. The server is temporarily overloading or down.
504	Gateway Timeout	The gateway has timed out.
505	HTTP Version Not Supported	The server does not support the **"http protocol"** version.

Methods to Set HTTP Status Code

Following methods can be used to set the HTTP Status Code in your servlet program. These methods are available with the *HttpServletResponse* object.

S.No.	Method & Description
1	**public void setStatus (int statusCode)** This method sets an arbitrary status code. The setStatus method takes an int (the status code) as an argument. If your response includes a special status code and a document, be sure to call **setStatus** before actually returning any of the content with the *PrintWriter*.

2	**public void sendRedirect(String url)** This method generates a 302 response along with a *Location* header giving the URL of the new document.
3	**public void sendError(int code, String message)** This method sends a status code (usually 404) along with a short message that is automatically formatted inside an HTML document and sent to the client.

HTTP Status Code Example

Following example shows how a 407 error code is sent to the client browser. After this, the browser would show you "**Need authentication!!!**" message.

```
<html>
   <head>
      <title>Setting HTTP Status Code</title>
   </head>

   <body>
      <%
         // Set error code and reason.
         response.sendError(407, "Need authentication!!!" );
      %>
   </body>
</html>
```

You will receive the following output −

```
HTTP Status 407 - Need authentication!!!

type Status report

message Need authentication!!!

description The client must first authenticate itself with the proxy (Need authentic

Apache Tomcat/5.5.29
```

To become more comfortable with HTTP status codes, try to set different status codes and their description.

JSP - Form Processing

In this chapter, we will discuss Form Processing in JSP. You must have come across many situations when you need to pass some information from your browser to the web server and ultimately to your backend program. The browser uses two methods to pass this information to the web server. These methods are the GET Method and the POST Method.

The Methods in Form Processing

Let us now discuss the methods in Form Processing.

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 the browser to the web server and it produces a long string that appears in your browser's **Location:box**. It is recommended that the GET method is better not used. 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 the GET method, 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

The following URL will pass two values to HelloForm program using the GET method.

http://localhost:8080/main.jsp?first_name=ZARA&last_name=ALI

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

```
<html>
   <head>
      <title>Using GET Method to Read Form Data</title>
   </head>

   <body>
      <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 will generate the following result −

Using GET Method to Read Form Data

First Name: ZARA

Last Name: ALI

GET Method Example Using Form

Following is an example that passes two values using the HTML FORM and the submit button. We are going to use the 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***, you will receive the following output.

First Name:

Last Name: Submit

< p>Try to enter the First Name and the Last Name and then click the 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 a little modification in the above JSP to handle both the GET and the POST method. Below is the **main.jsp** JSP program to handle the input given by web browser using the GET or the POST methods.

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

```
<html>
   <head>
      <title>Using GET and POST Method to Read Form Data</title>
   </head>
   
   <body>
      <center>
      <h1>Using POST 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 the **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>
```

Let us now keep **main.jsp** and hello.htm in **<Tomcat-installationdirectory>/webapps/ROOT directory**. When you access ***http://localhost:8080/Hello.htm***, you will receive the following output.

First Name:

Last Name: Submit

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

Based on the input provided, you will receive similar results as in the above examples.

Passing Checkbox Data to JSP Program

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

Following is an 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 above code will generate the following result −

[✓ Maths] Physics [✓ Chemistry] Select Subject

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

```
<html>
   <head>
      <title>Reading Checkbox Data</title>
   </head>

   <body>
      <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>
```

The above program will generate the following result −

```
Reading Checkbox Data

Maths Flag :: on

Physics Flag:: null

Chemistry Flag:: on
```

Reading All Form Parameters

Following is a 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 the ***hasMoreElements()*** method to determine when to stop and using the ***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 the **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 the 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 the above JSP to read any other form's data which is having other objects like text box, radio button or dropdown, etc.

JSP - Filters

In this chapter, we will discuss Filters in JSP. Servlet and JSP Filters are Java classes that can be used in Servlet and JSP Programming for the following purposes −

- To intercept requests from a client before they access a resource at back end.
- To manipulate responses from server before they are sent back to the client.

There are various types of filters suggested by the specifications −

- Authentication Filters
- Data compression Filters
- Encryption Filters
- Filters that trigger resource access events
- Image Conversion Filters
- Logging and Auditing Filters
- MIME-TYPE Chain Filters
- Tokenizing Filters
- XSL/T Filters That Transform XML Content

Filters are deployed in the deployment descriptor file **web.xml** and then map to either servlet or JSP names or URL patterns in your application's deployment descriptor. The deployment descriptor file web.xml can be found in *<Tomcat-installation-directory>\conf* directory.

When the JSP container starts up your web application, it creates an instance of each filter that you have declared in the deployment descriptor. The filters execute in the order that they are declared in the deployment descriptor.

Servlet Filter Methods

A filter is simply a Java class that implements the **javax.servlet.Filter** interface. The javax.servlet.Filter interface defines three methods −

S.No.	Method & Description
1	**public void doFilter (ServletRequest, ServletResponse, FilterChain)** This method is called by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain.
2	**public void init(FilterConfig filterConfig)** This method is called by the web container to indicate to a filter that it is being placed into service.
3	**public void destroy()** This method is called by the web container to indicate to a filter that it is being taken out of service.

JSP Filter Example

Following example shows how to print the client's IP address and the current date time, each time it would access any JSP file. This example will give you a basic understanding of the JSP Filter, but you can write more sophisticated filter applications using the same concept −

```
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// Implements Filter class
public class LogFilter implements Filter  {
   public void  init(FilterConfig config) throws ServletException {
      // Get init parameter
      String testParam = config.getInitParameter("test-param");

      //Print the init parameter
      System.out.println("Test Param: " + testParam);
   }
   public void  doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
      throws java.io.IOException, ServletException {

      // Get the IP address of client machine.
      String ipAddress = request.getRemoteAddr();
      // Log the IP address and current timestamp.
      System.out.println("IP "+ ipAddress + ", Time "+ new Date().toString());

      // Pass request back down the filter chain
      chain.doFilter(request,response);
   }
   public void destroy( ) {
      /* Called before the Filter instance is removed
      from service by the web container*/
   }
}
```

Compile **LogFilter.java** in the usual way and put your **LogFilter.class** file in **<Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes**.

JSP Filter Mapping in Web.xml

Filters are defined and then mapped to a URL or JSP file name, in much the same way as Servlet is defined and then mapped to a URL pattern in **web.xml** file. Create the following entry for filter tag in the deployment descriptor file **web.xml**

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogFilter</filter-class>

   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

The above filter will apply to all the servlets and JSP because we specified **/*** in our configuration. You can specify a particular servlet or the JSP path if you want to apply filter on few servlets or JSP only.

Now try to call any servlet or JSP and you will see generated log in you web server log. You can use **Log4J logger** to log above log in a separate file.

Using Multiple Filters

Your web application may define several different filters with a specific purpose. Consider, you define two filters *AuthenFilter* and *LogFilter*. Rest of the process will remain as explained above except you need to create a different mapping as mentioned below −

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter>
   <filter-name>AuthenFilter</filter-name>
   <filter-class>AuthenFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Filters Application Order

The order of filter-mapping elements in web.xml determines the order in which the web container applies the filter to the servlet or JSP. To reverse the order of the filter, you just need to reverse the filter-mapping elements in the **web.xml** file.

For example, the above example will apply the LogFilter first and then it will apply AuthenFilter to any servlet or JSP; the following example will reverse the order −

```
<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

JSP - Cookies Handling

In this chapter, we will discuss Cookies Handling in JSP. Cookies are text files stored on the client computer and they are kept for various information tracking purposes. JSP transparently supports HTTP cookies using underlying servlet technology.

There are three steps involved in identifying and returning users −

Server script sends a set of cookies to the browser. For example, name, age, or identification number, etc.

Browser stores this information on the local machine for future use.

When the next time the browser sends any request to the web server then it sends those cookies information to the server and server uses that information to identify the user or may be for some other purpose as well.

This chapter will teach you how to set or reset cookies, how to access them and how to delete them using JSP programs.

The Anatomy of a Cookie

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A JSP that sets a cookie might send headers that look something like this −

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name = xyz; expires = Friday, 04-Feb-07 22:03:38 GMT;
   path = /; domain = tutorialspoint.com
Connection: close
Content-Type: text/html
```

As you can see, the **Set-Cookie header** contains **a name value pair, a GMT date, a path** and **a domain**. The name and value will be URL encoded. The **expires** field is an instruction to the browser to **"forget"** the cookie after the given time and date.

If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server. The browser's headers might look something like this −

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126

Accept: image/gif, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: name = xyz
```

A JSP script will then have access to the cookies through the request method ***request.getCookies()*** which returns an array of *Cookie* objects.

Servlet Cookies Methods

Following table lists out the useful methods associated with the Cookie object which you can use while manipulating cookies in JSP −

S.No.	Method & Description
1	**public void setDomain(String pattern)** This method sets the domain to which the cookie applies; for example, tutorialspoint.com.
2	**public String getDomain()** This method gets the domain to which the cookie applies; for example, tutorialspoint.com.
3	**public void setMaxAge(int expiry)** This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
4	**public int getMaxAge()** This method returns the maximum age of the cookie, specified in seconds, By default, **-1** indicating the cookie will persist until the browser shutdown.
5	**public String getName()** This method returns the name of the cookie. The name cannot be changed after the creation.
6	**public void setValue(String newValue)** This method sets the value associated with the cookie.
7	**public String getValue()** This method gets the value associated with the cookie.
8	**public void setPath(String uri)** This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.
9	**public String getPath()**

	This method gets the path to which this cookie applies.
10	**public void setSecure(boolean flag)** This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e, SSL) connections.
11	**public void setComment(String purpose)** This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.
12	**public String getComment()** This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.

Setting Cookies with JSP

Setting cookies with JSP involves three steps −

Step 1: Creating a Cookie object

You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

```
Cookie cookie = new Cookie("key","value");
```

Keep in mind, neither the name nor the value should contain white space or any of the following characters −

```
[ ] ( ) = , " / ? @ : ;
```

Step 2: Setting the maximum age

You use **setMaxAge** to specify how long (in seconds) the cookie should be valid. The following code will set up a cookie for 24 hours.

```
cookie.setMaxAge(60*60*24);
```

Step 3: Sending the Cookie into the HTTP response headers

You use **response.addCookie** to add cookies in the HTTP response header as follows

```
response.addCookie(cookie);
```

Example

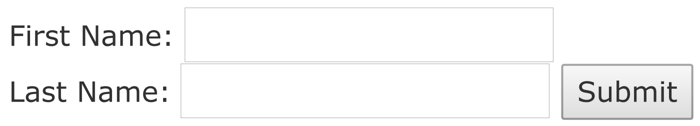
Let us modify our Form Example to set the cookies for the first and the last name.

```
<%
   // Create cookies for first and last names.
   Cookie firstName = new Cookie("first_name", request.getParameter("first_name"));
   Cookie lastName = new Cookie("last_name", request.getParameter("last_name"));

   // Set expiry date after 24 Hrs for both the cookies.
   firstName.setMaxAge(60*60*24);
   lastName.setMaxAge(60*60*24);

   // Add both the cookies in the response header.
   response.addCookie( firstName );
   response.addCookie( lastName );
%>

<html>
   <head>
      <title>Setting Cookies</title>
   </head>

   <body>
      <center>
         <h1>Setting Cookies</h1>
      </center>
      <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>
```

Let us put the above code in **main.jsp** file and use it in the following HTML page –

```
<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 the above HTML content in a file **hello.jsp** and put **hello.jsp** and **main.jsp** in **<Tomcat-installation-directory>/webapps/ROOT** directory. When you will access ***http://localhost:8080/hello.jsp***, here is the actual output of the above form.



First Name:

Last Name:

Try to enter the First Name and the Last Name and then click the submit button. This will display the first name and the last name on your screen and will also set two cookies **firstName** and **lastName**. These cookies will be passed back to the server when the next time you click the Submit button.

In the next section, we will explain how you can access these cookies back in your web application.

Reading Cookies with JSP

To read cookies, you need to create an array of *javax.servlet.http.Cookie* objects by calling the **getCookies( )** method of *HttpServletRequest*. Then cycle through the array, and use **getName()** and **getValue()** methods to access each cookie and associated value.

Example

Let us now read cookies that were set in the previous example –

```
<html>
   <head>
      <title>Reading Cookies</title>
   </head>

   <body>
      <center>
         <h1>Reading Cookies</h1>
      </center>
      <%
         Cookie cookie = null;
         Cookie[] cookies = null;

         // Get an array of Cookies associated with the this domain
         cookies = request.getCookies();

         if( cookies != null ) {
            out.println("<h2> Found Cookies Name and Value</h2>");

            for (int i = 0; i < cookies.length; i++) {
               cookie = cookies[i];
               out.print("Name : " + cookie.getName( ) + ",  ");
               out.print("Value: " + cookie.getValue( )+" <br/>");
            }
         } else {
            out.println("<h2>No cookies founds</h2>");
         }
      %>
   </body>

</html>
```

Let us now put the above code in **main.jsp** file and try to access it. If you set the **first_name cookie** as "John" and the **last_name cookie** as "Player" then running ***http://localhost:8080/main.jsp*** will display the following result –

```
Found Cookies Name and Value

Name : first_name, Value: John

Name : last_name,  Value: Player
```

Delete Cookies with JSP

To delete cookies is very simple. If you want to delete a cookie, then you simply need to follow these three steps –

Read an already existing cookie and store it in Cookie object.

Set cookie age as zero using the **setMaxAge()** method to delete an existing cookie.

Add this cookie back into the response header.

Example

Following example will show you how to delete an existing cookie named **"first_name"** and when you run main.jsp JSP next time, it will return null value for first_name.

```
<html>
   <head>
      <title>Reading Cookies</title>
   </head>
   
   <body>
      <center>
         <h1>Reading Cookies</h1>
      </center>
      <%
         Cookie cookie = null;
         Cookie[] cookies = null;
         
         // Get an array of Cookies associated with the this domain
         cookies = request.getCookies();
         
         if( cookies != null ) {
            out.println("<h2> Found Cookies Name and Value</h2>");
            
            for (int i = 0; i < cookies.length; i++) {
               cookie = cookies[i];
               if((cookie.getName( )).compareTo("first_name") == 0 ) {
                  cookie.setMaxAge(0);
                  response.addCookie(cookie);
                  out.print("Deleted cookie: " +
                  cookie.getName( ) + "<br/>");
               }
               out.print("Name : " + cookie.getName( ) + ",  ");
               out.print("Value: " + cookie.getValue( )+" <br/>");
            }
         } else {
            out.println(
            "<h2>No cookies founds</h2>");
         }
      %>
   </body>

</html>
```

Let us now put the above code in the **main.jsp** file and try to access it. It will display the following result –

```
Cookies Name and Value

Deleted cookie : first_name

Name : first_name, Value: John

Name : last_name,  Value: Player
```

Now run *http://localhost:8080/main.jsp* once again and it should display only one cookie as follows –

```
 Found Cookies Name and Value

Name : last_name,  Value: Player
```

You can delete your cookies in the Internet Explorer manually. Start at the Tools menu and select the Internet Options. To delete all cookies, click the Delete Cookies button.

JSP - Session Tracking

In this chapter, we will discuss session tracking in JSP. HTTP is a "stateless" protocol which means each time a client retrieves a Webpage, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.

Maintaining Session Between Web Client And Server

Let us now discuss a few options to maintain the session between the Web Client and the Web Server −

Cookies

A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.

This may not be an effective way as the browser at times does not support a cookie. It is not recommended to use this procedure to maintain the sessions.

Hidden Form Fields

A web server can send a hidden HTML form field along with a unique session ID as follows −

```
<input type = "hidden" name = "sessionid" value = "12345">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the **GET** or the **POST** data. Each time the web browser sends the request back, the **session_id** value can be used to keep the track of different web browsers.

This can be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

URL Rewriting

You can append some extra data at the end of each URL. This data identifies the session; the server can associate that session identifier with the data it has stored about that session.

For example, with **http://tutorialspoint.com/file.htm;sessionid=12345**, the session identifier is attached as **sessionid = 12345** which can be accessed at the web server to identify the client.

URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies. The drawback here is that you will have to generate every URL dynamically to assign a session ID though page is a simple static HTML page.

The session Object

Apart from the above mentioned options, JSP makes use of the servlet provided HttpSession Interface. This interface provides a way to identify a user across.

- a one page request or
- visit to a website or
- store information about that user

By default, JSPs have session tracking enabled and a new HttpSession object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows −

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

The JSP engine exposes the HttpSession object to the JSP author through the implicit **session** object. Since **session** object is already provided to the JSP programmer, the programmer can immediately begin storing and retrieving data from the object without any initialization or **getSession()**.

Here is a summary of important methods available through the session object −

S.No.	Method & Description
1	**public Object getAttribute(String name)** This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	**public Enumeration getAttributeNames()** This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3	**public long getCreationTime()** This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4	**public String getId()** This method returns a string containing the unique identifier assigned to this session.
5	**public long getLastAccessedTime()** This method returns the last time the client sent a request associated with the this session, as the number of milliseconds since midnight January 1, 1970

	GMT.
6	**public int getMaxInactiveInterval()** This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
7	**public void invalidate()** This method invalidates this session and unbinds any objects bound to it.
8	**public boolean isNew()** This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
9	**public void removeAttribute(String name)** This method removes the object bound with the specified name from this session.
10	**public void setAttribute(String name, Object value)** This method binds an object to this session, using the name specified.
11	**public void setMaxInactiveInterval(int interval)** This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

Session Tracking Example

This example describes how to use the HttpSession object to find out the creation time and the last-accessed time for a session. We would associate a new session with the request if one does not already exist.

```
<%@ page import = "java.io.*,java.util.*" %>
<%
   // Get session creation time.
   Date createTime = new Date(session.getCreationTime());

   // Get last access time of this Webpage.
   Date lastAccessTime = new Date(session.getLastAccessedTime());

   String title = "Welcome Back to my website";
   Integer visitCount = new Integer(0);
   String visitCountKey = new String("visitCount");
   String userIDKey = new String("userID");
   String userID = new String("ABCD");

   // Check if this is new comer on your Webpage.
   if (session.isNew() ){
      title = "Welcome to my website";
      session.setAttribute(userIDKey, userID);
      session.setAttribute(visitCountKey,  visitCount);
   }
   visitCount = (Integer)session.getAttribute(visitCountKey);
   visitCount = visitCount + 1;
   userID = (String)session.getAttribute(userIDKey);
   session.setAttribute(visitCountKey,  visitCount);
%>

<html>
   <head>
      <title>Session Tracking</title>
   </head>

   <body>
      <center>
         <h1>Session Tracking</h1>
      </center>

      <table border = "1" align = "center">
         <tr bgcolor = "#949494">
            <th>Session info</th>
            <th>Value</th>
         </tr>
         <tr>
            <td>id</td>
            <td><% out.print( session.getId()); %></td>
         </tr>
         <tr>
            <td>Creation Time</td>
            <td><% out.print(createTime); %></td>
         </tr>
         <tr>
            <td>Time of Last Access</td>
            <td><% out.print(lastAccessTime); %></td>
         </tr>
         <tr>
            <td>User ID</td>
            <td><% out.print(userID); %></td>
         </tr>
         <tr>
            <td>Number of visits</td>
            <td><% out.print(visitCount); %></td>
         </tr>
      </table>

   </body>
</html>
```

Now put the above code in **main.jsp** and try to access ***http://localhost:8080/main.jsp***. Once you run the URL, you will receive the following result −

Welcome to my website

Session Information

Session info	value
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	0

Now try to run the same JSP for the second time, you will receive the following result.

Welcome Back to my website

Session Information

info type	value
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	1

Deleting Session Data

When you are done with a user's session data, you have several options −

Remove a particular attribute − You can call the ***public void removeAttribute(String name)*** method to delete the value associated with the a particular key.

Delete the whole session − You can call the ***public void invalidate()*** method to discard an entire session.

Setting Session timeout − You can call the ***public void setMaxInactiveInterval(int interval)*** method to set the timeout for a session individually.

Log the user out − The servers that support servlets 2.4, you can call **logout** to log the client out of the Web server and invalidate all sessions belonging to all the users.

web.xml Configuration – If you are using Tomcat, apart from the above mentioned methods, you can configure the session time out in web.xml file as follows.

```
<session-config>
   <session-timeout>15</session-timeout>
</session-config>
```

The timeout is expressed as minutes, and overrides the default timeout which is 30 minutes in Tomcat.

The **getMaxInactiveInterval( )** method in a servlet returns the timeout period for that session in seconds. So if your session is configured in web.xml for 15 minutes, **getMaxInactiveInterval( )** returns 900.

JSP - File Uploading

In this chapter, we will discuss File Uploading in JSP. A JSP can be used with an HTML form tag to allow users to upload files to the server. An uploaded file can be a text file or a binary or an image file or just any document.

Creating a File Upload Form

Let us now understand how to create a file upload form. The following HTML code creates an uploader form. Following are the important points to be noted down –

The form **method** attribute should be set to **POST** method and GET method can not be used.

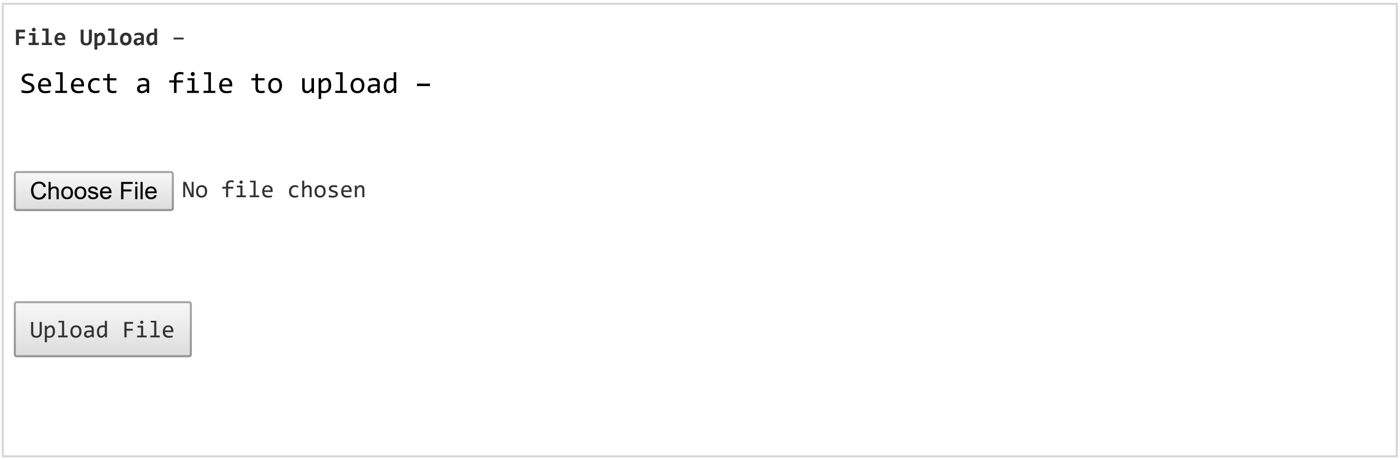
The form **enctype** attribute should be set to **multipart/form-data**.

The form **action** attribute should be set to a JSP file which would handle file uploading at backend server. Following example is using **uploadFile.jsp** program file to upload file.

To upload a single file you should use a single **<input .../>** tag with attribute **type = "file"**. To allow multiple files uploading, include more than one input tag with different values for the name attribute. The browser associates a Browse button with each of them.

```
<html>
   <head>
      <title>File Uploading Form</title>
   </head>

   <body>
      <h3>File Upload:</h3>
      Select a file to upload: <br />
      <form action = "UploadServlet" method = "post"
         enctype = "multipart/form-data">
         <input type = "file" name = "file" size = "50" />
         <br />
         <input type = "submit" value = "Upload File" />
      </form>
   </body>

</html>
```

This will display the following result. You can now select a file from the local PC and when the user clicks "Upload File", the form gets submitted along with the selected file –


File Upload –

Select a file to upload –

NOTE – Above form is just dummy form and would not work, you should try above code at your machine to make it work.

Writing Backend JSP Script

Let us now define a location where the uploaded files will be stored. You can hard code this in your program or this directory name can also be added using an external configuration such as a **context-param** element in web.xml as follows –

```
<web-app>
....
<context-param>
   <description>Location to store uploaded file</description>
   <param-name>file-upload</param-name>
   <param-value>
      c:\apache-tomcat-5.5.29\webapps\data\
   </param-value>
</context-param>
....
</web-app>
```

Following is the source code for **UploadFile.jsp**. This can handle uploading of multiple files at a time. Let us now consider the following before proceeding with the uploading of files.

> The following example depends on **FileUpload**; make sure you have the latest version of **commons-fileupload.x.x.jar** file in your classpath. You can download

it from https://commons.apache.org/fileupload/ .

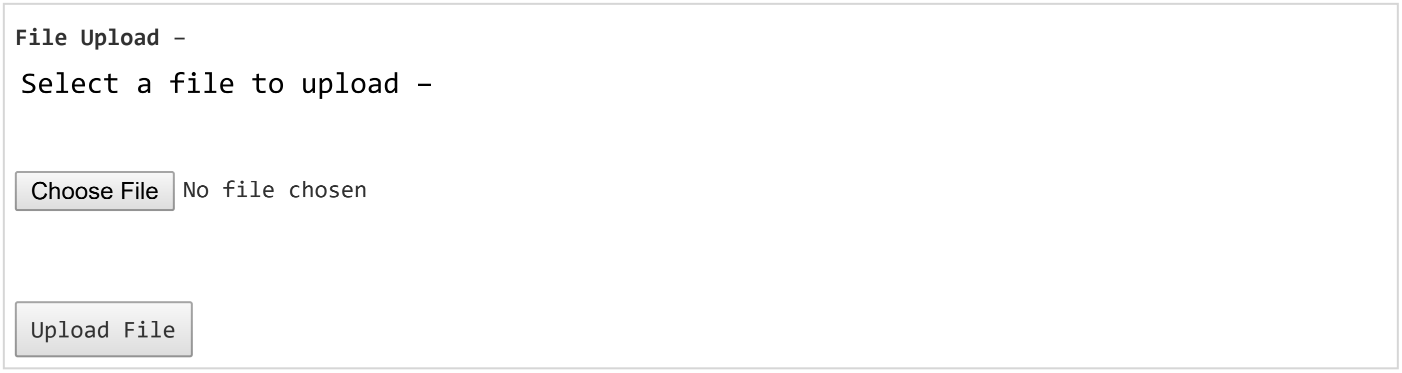
FileUpload depends on Commons IO; make sure you have the latest version of **commons-io-x.x.jar** file in your classpath. You can download it from https://commons.apache.org/io/ .

While testing the following example, you should upload a file which is of less size than *maxFileSize* otherwise the file will not be uploaded.

Make sure you have created directories **c:\temp** and **c:\apache-tomcat5.5.29\webapps\data** well in advance.

```
<%@ page import = "java.io.*,java.util.*, javax.servlet.*" %>
<%@ page import = "javax.servlet.http.*" %>
<%@ page import = "org.apache.commons.fileupload.*" %>
<%@ page import = "org.apache.commons.fileupload.disk.*" %>
<%@ page import = "org.apache.commons.fileupload.servlet.*" %>
<%@ page import = "org.apache.commons.io.output.*" %>

<%
   File file ;
   int maxFileSize = 5000 * 1024;
   int maxMemSize = 5000 * 1024;
   ServletContext context = pageContext.getServletContext();
   String filePath = context.getInitParameter("file-upload");

   // Verify the content type
   String contentType = request.getContentType();

   if ((contentType.indexOf("multipart/form-data") >= 0)) {
      DiskFileItemFactory factory = new DiskFileItemFactory();
      // maximum size that will be stored in memory
      factory.setSizeThreshold(maxMemSize);

      // Location to save data that is larger than maxMemSize.
      factory.setRepository(new File("c:\\temp"));

      // Create a new file upload handler
      ServletFileUpload upload = new ServletFileUpload(factory);

      // maximum file size to be uploaded.
      upload.setSizeMax( maxFileSize );

      try {
         // Parse the request to get file items.
         List fileItems = upload.parseRequest(request);

         // Process the uploaded file items
         Iterator i = fileItems.iterator();

         out.println("<html>");
         out.println("<head>");
         out.println("<title>JSP File upload</title>");
         out.println("</head>");
         out.println("<body>");

         while ( i.hasNext () ) {
            FileItem fi = (FileItem)i.next();
            if ( !fi.isFormField () ) {
               // Get the uploaded file parameters
               String fieldName = fi.getFieldName();
               String fileName = fi.getName();
               boolean isInMemory = fi.isInMemory();
               long sizeInBytes = fi.getSize();

               // Write the file
               if( fileName.lastIndexOf("\\") >= 0 ) {
                  file = new File( filePath +
                  fileName.substring( fileName.lastIndexOf("\\"))) ;
               } else {
                  file = new File( filePath +
                  fileName.substring(fileName.lastIndexOf("\\")+1)) ;
               }
               fi.write( file ) ;
               out.println("Uploaded Filename: " + filePath +
               fileName + "<br>");
            }
         }
         out.println("</body>");
         out.println("</html>");
      } catch(Exception ex) {
         System.out.println(ex);
      }
   } else {
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Servlet upload</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<p>No file uploaded</p>");
      out.println("</body>");
      out.println("</html>");
   }
%>
```

Now try to upload files using the HTML form which you created above. When you try **http://localhost:8080/UploadFile.htm**, it will display the following result. This will help you upload any file from your local machine.



File Upload -

Select a file to upload -

If your JSP script works fine, your file should be uploaded in **c:\apache-tomcat5.5.29\webapps\data\** directory.

JSP - Handling Date

In this chapter, we will discuss how to handle data in JSP. One of the most important advantages of using JSP is that you can use all the methods available in core Java. We will take you through the **Date** class which is available in the **java.util** package; this class encapsulates the current date and time.

The Date class supports two constructors. The first constructor initializes the object with the current date and time.

```
Date( )
```

The following constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

```
Date(long millisec)
```

Once you have a Date object available, you can call any of the following support methods to play with dates −

S.No.	Methods & Description
1	**boolean after(Date date)** Returns true if the invoking Date object contains a date that is later than the one specified by date, otherwise, it returns false.
2	**boolean before(Date date)** Returns true if the invoking Date object contains a date that is earlier than the one specified by date, otherwise, it returns false.
3	**Object clone( )** Duplicates the invoking Date object.
4	**int compareTo(Date date)** Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than date. Returns a positive value if the invoking object is later than date.
5	**int compareTo(Object obj)** Operates identically to compareTo(Date) if obj is of class Date. Otherwise, it throws a ClassCastException.
6	**boolean equals(Object date)**

	Returns true if the invoking Date object contains the same time and date as the one specified by date, otherwise, it returns false.
7	**long getTime( )** Returns the number of milliseconds that have elapsed since January 1, 1970.
8	**int hashCode( )** Returns a hash code for the invoking object.
9	**void setTime(long time)** Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1, 1970
10	**String toString( )** Converts the invoking Date object into a string and returns the result.

Getting Current Date and Time

With JSP program, it is very easy to get the current date and the time. You can use a simple Date object with the ***toString()*** method to print the current date and the time as follows −

```
<%@ page import = "java.io.*,java.util.*, javax.servlet.*" %>

<html>
   <head>
      <title>Display Current Date & Time</title>
   </head>
   
   <body>
      <center>
         <h1>Display Current Date & Time</h1>
      </center>
      <%
         Date date = new Date();
         out.print( "<h2 align = \"center\">" +date.toString()+"</h2>");
      %>
   </body>
</html>
```

Let us now keep the code in **CurrentDate.jsp** and then call this JSP using the URL **http://localhost:8080/CurrentDate.jsp**. You will receive the following result −

Display Current Date & Time

```
Mon Jun 21 21:46:49 GMT+04:00 2010
```

Refresh the page with the **URL http://localhost:8080/CurrentDate.jsp**. You will find difference in seconds everytime you would refresh.

Date Comparison

As discussed in the previous sections, you can use all the available Java methods in your JSP scripts. In case you need to compare two dates, consider the following methods −

- You can use **getTime( )** method to obtain the number of milliseconds that have elapsed since midnight, January 1, 1970, for both objects and then compare these two values.
- You can use the methods **before( ), after( )**, and **equals( )** because the 12th of the month comes before the 18th; for example, **new Date(99, 2, 12).before(new Date (99, 2, 18))** returns true.
- You can use the **compareTo( )** method; this method is defined by the **Comparable interface** and implemented by Date.

Date Formatting using SimpleDateFormat

SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner. SimpleDateFormat allows you to start by choosing any user-defined patterns for date-time formatting.

Let us modify the above example as follows −

```
<%@ page import = "java.io.*,java.util.*" %>
<%@ page import = "javax.servlet.*,java.text.*" %>

<html>
   <head>
      <title>Display Current Date & Time</title>
   </head>

   <body>
      <center>
         <h1>Display Current Date & Time</h1>
      </center>
      <%
         Date dNow = new Date( );
         SimpleDateFormat ft =
         new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
         out.print( "<h2 align=\"center\">" + ft.format(dNow) + "</h2>");
      %>
   </body>
</html>
```

Compile the above servlet once again and then call this servlet using the URL **http://localhost:8080/CurrentDate**. You will receive the following result −

```
Display Current Date & Time

Mon 2010.06.21 at 10:06:44 PM GMT+04:00
```

Simple DateFormat Format Codes

To specify the time format, use a time pattern string. In this pattern, all ASCII letters are reserved as pattern letters, which are defined as the following −

Character	Description	Example
G	Era designator	AD
y	Year in four digits	2001
M	Month in year	July or 07
d	Day in month	0
h	Hour in A.M./P.M. (1~12)	2
H	Hour in day (0~23)	22
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	234
E	Day in week	Tuesday
D	Day in year	360
F	Day of week in month	2 (second Wed. in July)
w	Week in year	40
W	Week in month	
a	A.M./P.M. marker	PM

k	Hour in day (1~24)	24
K	Hour in A.M./P.M. (0~11)	0
z	Time zone	Eastern Standard Time
'	Escape for text	Delimiter
"	Single quote	`

For a complete list of constant available methods to manipulate date, you can refer to the standard Java documentation.

JSP - Page Redirecting

In this chapter, we will discuss page redirecting with JSP. Page redirection is generally used when a document moves to a new location and we need to send the client to this new location. This can be because of load balancing, or for simple randomization.

The simplest way of redirecting a request to another page is by using **sendRedirect()** method of response object. Following is the signature of this method −

```
public void response.sendRedirect(String location)
throws IOException
```

This method sends back the response to the browser along with the status code and new page location. You can also use the **setStatus()** and the **setHeader()** methods together to achieve the same redirection example −

```
....
String site = "http://www.newpage.com" ;
response.setStatus(response.SC_MOVED_TEMPORARILY);
response.setHeader("Location", site);
....
```

Example

This example shows how a JSP performs page redirection to an another location −

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

<html>
   <head>
      <title>Page Redirection</title>
   </head>

   <body>
      <center>
         <h1>Page Redirection</h1>
      </center>
      <%
         // New location to be redirected
         String site = new String("http://www.photofuntoos.com");
         response.setStatus(response.SC_MOVED_TEMPORARILY);
         response.setHeader("Location", site);
      %>
   </body>
</html>
```

Let us now put the above code in PageRedirect.jsp and call this JSP using the URL **http://localhost:8080/PageRedirect.jsp**. This would take you to the given URL **http://www.photofuntoos.com**.

JSP - Hits Counter

In this chapter, we will discuss Hits Counter in JSP. A hit counter tells you about the number of visits on a particular page of your web site. Usually you attach a hit counter with your index.jsp page assuming people first land on your home page.

To implement a hit counter you can make use of the Application Implicit object and associated methods **getAttribute()** and **setAttribute()**.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the **jspDestroy()** method.

Following is the syntax to set a variable at application level –

```
application.setAttribute(String Key, Object Value);
```

You can use the above method to set a hit counter variable and to reset the same variable. Following is the method to read the variable set by the previous method –

```
application.getAttribute(String Key);
```

Every time a user accesses your page, you can read the current value of the hit counter and increase it by one and again set it for future use.

Example

This example shows how you can use JSP to count the total number of hits on a particular page. If you want to count the total number of hits of your website then you will have to include the same code in all the JSP pages.

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

<html>
   <head>
      <title>Application object in JSP</title>
   </head>

   <body>
      <%
```

```
         Integer hitsCount = (Integer)application.getAttribute("hitCounter");
         if( hitsCount ==null || hitsCount == 0 ) {
            /* First visit */
            out.println("Welcome to my website!");
            hitsCount = 1;
         } else {
            /* return visit */
            out.println("Welcome back to my website!");
            hitsCount += 1;
         }
         application.setAttribute("hitCounter", hitsCount);
      %>
      <center>
         <p>Total number of visits: <%= hitsCount%></p>
      </center>

   </body>
</html>
```

Let us now put the above code in **main.jsp** and call this JSP using the URL **http://localhost:8080/main.jsp**. This will display the hit counter value which increases as and when you refresh the page. You can try accessing the page using different browsers and you will find that the hit counter will keep increasing with every hit and you will receive the result as follows −

```

Welcome back to my website!


Total number of visits: 12

```

Hit Counter Resets

What when you restart your application, i.e., web server, this will reset your application variable and your counter will reset to zero. To avoid this loss, consider the following points −

- Define a database table with a single count, let us say hitcount. Assign a zero value to it.
- With every hit, read the table to get the value of hitcount.
- Increase the value of hitcount by one and update the table with new value.
- Display new value of hitcount as total page hit counts.
- If you want to count hits for all the pages, implement above logic for all the pages.

JSP - Auto Refresh

In this chapter, we will discuss Auto Refresh in JSP. Consider a webpage which is displaying live game score or stock market status or currency exchange ration. For all such type of pages, you would need to refresh your Webpage regularly using refresh or reload button with your browser.

JSP makes this job easy by providing you a mechanism where you can make a webpage in such a way that it would refresh automatically after a given interval.

The simplest way of refreshing a Webpage is by using the **setIntHeader()** method of the response object. Following is the signature of this method −

```
public void setIntHeader(String header, int headerValue)
```

This method sends back the header "Refresh" to the browser along with an integer value which indicates time interval in seconds.

Auto Page Refresh Example

In the following example, we will use the **setIntHeader()** method to set **Refresh** header. This will help 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("Crrent Time: " + CT + "\n");
         %>
      </center>

   </body>
</html>
```

Now put the above code in **main.jsp** and try to access it. This will display the 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
```

JSP - Sending Email

In this chapter, we will discuss how to send emails using JSP. To send an email using a JSP, you should have the **JavaMail API** and the **Java Activation Framework (JAF)** installed on your machine.

> You can download the latest version of JavaMail (Version 1.2) from the Java's standard website.
>
> You can download the latest version of JavaBeans Activation Framework JAF (Version 1.0.2) from the Java's standard website.

Download and unzip these files, in the newly-created top-level directories. You will find a number of jar files for both the applications. You need to add the **mail.jar** and the **activation.jar** files in your CLASSPATH.

Send a Simple Email

Here is an example to send a simple email from your machine. It is assumed that your **localhost** is connected to the Internet and that it is capable enough to send an email. Make sure all the jar files from the Java Email API package and the JAF package are available in CLASSPATH.

```
<%@ page import = "java.io.*,java.util.*,javax.mail.*"%>
<%@ page import = "javax.mail.internet.*,javax.activation.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>

<%
   String result;

   // Recipient's email ID needs to be mentioned.
   String to = "abcd@gmail.com";

   // Sender's email ID needs to be mentioned
   String from = "mcmohd@gmail.com";

   // Assuming you are sending email from localhost
   String host = "localhost";

   // Get system properties object
   Properties properties = System.getProperties();
   // Setup mail server
   properties.setProperty("mail.smtp.host", host);

   // Get the default Session object.
   Session mailSession = Session.getDefaultInstance(properties);

   try {
      // Create a default MimeMessage object.
      MimeMessage message = new MimeMessage(mailSession);

      // Set From: header field of the header.
      message.setFrom(new InternetAddress(from));

      // Set To: header field of the header.
      message.addRecipient(Message.RecipientType.TO,
                               new InternetAddress(to));
      // Set Subject: header field
      message.setSubject("This is the Subject Line!");

      // Now set the actual message
      message.setText("This is actual message");

      // Send message
      Transport.send(message);
      result = "Sent message successfully....";
   } catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
   }
%>

<html>
   <head>
      <title>Send Email using JSP</title>
   </head>

   <body>
      <center>
         <h1>Send Email using JSP</h1>
      </center>

      <p align = "center">
         <%
            out.println("Result: " + result + "\n");
         %>
      </p>
   </body>
</html>
```

Let us now put the above code in **SendEmail.jsp** file and call this JSP using the URL **http://localhost:8080/SendEmail.jsp**. This will help send an email to the given email ID **abcd@gmail.com**. You will receive the following response –

```
Send Email using JSP
```

```
Result: Sent message successfully....
```

If you want to send an email to multiple recipients, then use the following methods to specify multiple email IDs −

```
void addRecipients(Message.RecipientType type, Address[] addresses)
throws MessagingException
```

Here is the description of the parameters −

> **type** − This would be set to TO, CC or BCC. Here CC represents Carbon Copy and BCC represents Black Carbon Copy. Example *Message.RecipientType.TO*
>
> **addresses** − This is the array of email ID. You would need to use the InternetAddress() method while specifying email IDs

Send an HTML Email

Here is an example to send an HTML email from your machine. It is assumed that your **localhost** is connected to the Internet and that it is capable enough to send an email. Make sure all the jar files from the **Java Email API package** and the **JAF package** are available in CLASSPATH.

This example is very similar to the previous one, except that here we are using the **setContent()** method to set content whose second argument is **"text/html"** to specify that the HTML content is included in the message.

Using this example, you can send as big an HTML content as you require.

```
<%@ page import = "java.io.*,java.util.*,javax.mail.*"%>
<%@ page import = "javax.mail.internet.*,javax.activation.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>

<%
   String result;

   // Recipient's email ID needs to be mentioned.
   String to = "abcd@gmail.com";

   // Sender's email ID needs to be mentioned
   String from = "mcmohd@gmail.com";

   // Assuming you are sending email from localhost
   String host = "localhost";

   // Get system properties object
   Properties properties = System.getProperties();

   // Setup mail server
   properties.setProperty("mail.smtp.host", host);
   // Get the default Session object.
   Session mailSession = Session.getDefaultInstance(properties);

   try {
      // Create a default MimeMessage object.
      MimeMessage message = new MimeMessage(mailSession);

      // Set From: header field of the header.
      message.setFrom(new InternetAddress(from));

      // Set To: header field of the header.
      message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

      // Set Subject: header field
      message.setSubject("This is the Subject Line!");

      // Send the actual HTML message, as big as you like
      message.setContent("<h1>This is actual message</h1>", "text/html" );

      // Send message
      Transport.send(message);
      result = "Sent message successfully....";
   } catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
   }
%>

<html>
   <head>
      <title>Send HTML Email using JSP</title>
   </head>

   <body>
      <center>
         <h1>Send Email using JSP</h1>
      </center>

      <p align = "center">
         <%
            out.println("Result: " + result + "\n");
         %>
      </p>
   </body>
</html>
```

Let us now use the above JSP to send HTML message on a given email ID.

Send Attachment in Email

Following is an example to send an email with attachment from your machine –

```
<%@ page import = "java.io.*,java.util.*,javax.mail.*"%>
<%@ page import = "javax.mail.internet.*,javax.activation.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>

<%
   String result;
```

```
// Recipient's email ID needs to be mentioned.
String to = "abcd@gmail.com";

// Sender's email ID needs to be mentioned
String from = "mcmohd@gmail.com";

// Assuming you are sending email from localhost
String host = "localhost";

// Get system properties object
Properties properties = System.getProperties();

// Setup mail server
properties.setProperty("mail.smtp.host", host);

// Get the default Session object.
Session mailSession = Session.getDefaultInstance(properties);

try {
   // Create a default MimeMessage object.
   MimeMessage message = new MimeMessage(mailSession);

   // Set From: header field of the header.
   message.setFrom(new InternetAddress(from));

   // Set To: header field of the header.
   message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

   // Set Subject: header field
   message.setSubject("This is the Subject Line!");

   // Create the message part
   BodyPart messageBodyPart = new MimeBodyPart();

   // Fill the message
   messageBodyPart.setText("This is message body");

   // Create a multipart message
   Multipart multipart = new MimeMultipart();

   // Set text message part
   multipart.addBodyPart(messageBodyPart);

   // Part two is attachment
   messageBodyPart = new MimeBodyPart();

   String filename = "file.txt";
   DataSource source = new FileDataSource(filename);
   messageBodyPart.setDataHandler(new DataHandler(source));
   messageBodyPart.setFileName(filename);
   multipart.addBodyPart(messageBodyPart);

   // Send the complete message parts
   message.setContent(multipart );

   // Send message
   Transport.send(message);
   String title = "Send Email";
   result = "Sent message successfully....";
} catch (MessagingException mex) {
   mex.printStackTrace();
```

```
      result = "Error: unable to send message....";
   }
%>

<html>
   <head>
      <title>Send Attachment Email using JSP</title>
   </head>

   <body>
      <center>
         <h1>Send Attachment Email using JSP</h1>
      </center>

      <p align = "center">
         <%out.println("Result: " + result + "\n");%>
      </p>
   </body>
</html>
```

Let us now run the above JSP to send a file as an attachment along with a message on a given email ID.

User Authentication Part

If it is required to provide user ID and Password to the email server for authentication purpose, then you can set these properties as follows −

```
props.setProperty("mail.user", "myuser");

props.setProperty("mail.password", "mypwd");
```

Rest of the email sending mechanism will remain as explained above.

Using Forms to Send Email

You can use HTML form to accept email parameters and then you can use the **request** object to get all the information as follows −

```
String to = request.getParameter("to");
String from = request.getParameter("from");
String subject = request.getParameter("subject");
String messageText = request.getParameter("body");
```

Once you have all the information, you can use the above mentioned programs to send email.

JSP - Standard Tag Library (JSTL) Tutorial

In this chapter, we will understand the different tags in JSP. The JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates the core functionality common to many JSP applications.

JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating the existing custom tags with the JSTL tags.

Install JSTL Library

To begin working with JSP tages you need to first install the JSTL library. If you are using the Apache Tomcat container, then follow these two steps −

Step 1 − Download the binary distribution from Apache Standard Taglib and unpack the compressed file.

Step 2 − To use the Standard Taglib from its **Jakarta Taglibs distribution**, simply copy the JAR files in the distribution's 'lib' directory to your application's **webapps\ROOT\WEB-INF\lib** directory.

To use any of the libraries, you must include a <taglib> directive at the top of each JSP that uses the library.

Classification of The JSTL Tags

The JSTL tags can be classified, according to their functions, into the following JSTL tag library groups that can be used when creating a JSP page −

- **Core Tags**
- **Formatting tags**
- **SQL tags**
- **XML tags**
- **JSTL Functions**

Core Tags

The core group of tags are the most commonly used JSTL tags. Following is the syntax to include the JSTL Core library in your JSP −

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

Following table lists out the core JSTL Tags −

S.No.	Tag & Description
1	**<c:out>** Like <%= ... >, but for expressions.

2	**<c:set >** Sets the result of an expression evaluation in a **'scope'**
3	**<c:remove >** Removes a **scoped variable** (from a particular scope, if specified).
4	**<c:catch>** Catches any **Throwable** that occurs in its body and optionally exposes it.
5	**<c:if>** Simple conditional tag which evalutes its body if the supplied condition is true.
6	**<c:choose>** Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by **<when>** and **<otherwise>**.
7	**<c:when>** Subtag of **<choose>** that includes its body if its condition evalutes to **'true'**.
8	**<c:otherwise >** Subtag of **<choose>** that follows the **<when>** tags and runs only if all of the prior conditions evaluated to **'false'**.
9	**<c:import>** Retrieves an absolute or relative URL and exposes its contents to either the page, a String in **'var'**, or a Reader in **'varReader'**.
10	**<c:forEach >** The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality .
11	**<c:forTokens>** Iterates over tokens, separated by the supplied delimeters.
12	**<c:param>** Adds a parameter to a containing **'import'** tag's URL.
13	**<c:redirect >** Redirects to a new URL.
14	**<c:url>** Creates a URL with optional query parameters

Formatting Tags

The JSTL formatting tags are used to format and display text, the date, the time, and numbers for internationalized Websites. Following is the syntax to include Formatting library in your JSP −

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

Following table lists out the Formatting JSTL Tags −

S.No.	Tag & Description
1	**<fmt:formatNumber>** To render numerical value with specific precision or format.
2	**<fmt:parseNumber>** Parses the string representation of a number, currency, or percentage.
3	**<fmt:formatDate>** Formats a date and/or time using the supplied styles and pattern.
4	**<fmt:parseDate>** Parses the string representation of a date and/or time
5	**<fmt:bundle>** Loads a resource bundle to be used by its tag body.
6	**<fmt:setLocale>** Stores the given locale in the locale configuration variable.
7	**<fmt:setBundle>** Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable.
8	**<fmt:timeZone>** Specifies the time zone for any time formatting or parsing actions nested in its body.
9	**<fmt:setTimeZone>** Stores the given time zone in the time zone configuration variable

10	**<fmt:message>** Displays an internationalized message.
11	**<fmt:requestEncoding>** Sets the request character encoding

SQL Tags

The JSTL SQL tag library provides tags for interacting with relational databases (RDBMSs) such as **Oracle, mySQL**, or **Microsoft SQL Server**.

Following is the syntax to include JSTL SQL library in your JSP −

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

Following table lists out the SQL JSTL Tags −

S.No.	Tag & Description
1	**<sql:setDataSource>** Creates a simple DataSource suitable only for prototyping
2	**<sql:query>** Executes the SQL query defined in its body or through the sql attribute.
3	**<sql:update>** Executes the SQL update defined in its body or through the sql attribute.
4	**<sql:param>** Sets a parameter in an SQL statement to the specified value.
5	**<sql:dateParam>** Sets a parameter in an SQL statement to the specified java.util.Date value.
6	**<sql:transaction >** Provides nested database action elements with a shared Connection, set up to execute all statements as one transaction.

XML tags

The JSTL XML tags provide a JSP-centric way of creating and manipulating the XML documents. Following is the syntax to include the JSTL XML library in your JSP.

The JSTL XML tag library has custom tags for interacting with the XML data. This includes parsing the XML, transforming the XML data, and the flow control based on the XPath expressions.

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

Before you proceed with the examples, you will need to copy the following two XML and XPath related libraries into your **<Tomcat Installation Directory>\lib** –

XercesImpl.jar – Download it from https://www.apache.org/dist/xerces/j/

xalan.jar – Download it from https://xml.apache.org/xalan-j/index.html

Following is the list of XML JSTL Tags –

S.No.	Tag & Description
1	**<x:out>** Like <%= ... >, but for XPath expressions.
2	**<x:parse>** Used to parse the XML data specified either via an attribute or in the tag body.
3	**<x:set >** Sets a variable to the value of an XPath expression.
4	**<x:if >** Evaluates a test XPath expression and if it is true, it processes its body. If the test condition is false, the body is ignored.
5	**<x:forEach>** To loop over nodes in an XML document.
6	**<x:choose>** Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by **<when>** and **<otherwise>** tags.
7	**<x:when >** Subtag of **<choose>** that includes its body if its expression evalutes to 'true'.
8	**<x:otherwise >** Subtag of **<choose>** that follows the **<when>** tags and runs only if all of the prior conditions evaluates to 'false'.

9	**<x:transform >** Applies an XSL transformation on a XML document
10	**<x:param >** Used along with the **transform** tag to set a parameter in the XSLT stylesheet

JSTL Functions

JSTL includes a number of standard functions, most of which are common string manipulation functions. Following is the syntax to include JSTL Functions library in your JSP −

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

Following table lists out the various JSTL Functions −

S.No.	Function & Description
1	**fn:contains()** Tests if an input string contains the specified substring.
2	**fn:containsIgnoreCase()** Tests if an input string contains the specified substring in a case insensitive way.
3	**fn:endsWith()** Tests if an input string ends with the specified suffix.
4	**fn:escapeXml()** Escapes characters that can be interpreted as XML markup.
5	**fn:indexOf()** Returns the index withing a string of the first occurrence of a specified substring.
6	**fn:join()** Joins all elements of an array into a string.
7	**fn:length()** Returns the number of items in a collection, or the number of characters in a string.

8	**fn:replace()** Returns a string resulting from replacing in an input string all occurrences with a given string.
9	**fn:split()** Splits a string into an array of substrings.
10	**fn:startsWith()** Tests if an input string starts with the specified prefix.
11	**fn:substring()** Returns a subset of a string.
12	**fn:substringAfter()** Returns a subset of a string following a specific substring.
13	**fn:substringBefore()** Returns a subset of a string before a specific substring.
14	**fn:toLowerCase()** Converts all of the characters of a string to lower case.
15	**fn:toUpperCase()** Converts all of the characters of a string to upper case.
16	**fn:trim()** Removes white spaces from both ends of a string.

JSP - Database Access

In this chapter, we will discuss how to access database with JSP. We assume you have good understanding on how JDBC application works. Before starting with database access through a JSP, make sure you have proper JDBC environment setup along with a database.

For more detail on how to access database using JDBC and its environment setup you can go through our JDBC Tutorial .

To start with basic concept, let us create a table and create a few records in that table as follows −

Create Table

To create the **Employees** table in the EMP database, use the following steps −

Step 1

Open a **Command Prompt** and change to the installation directory as follows −

```
C:\>
C:\>cd Program Files\MySQL\bin
C:\Program Files\MySQL\bin>
```

Step 2

Login to the database as follows −

```
C:\Program Files\MySQL\bin>mysql -u root -p
Enter password: ********
mysql>
```

Step 3

Create the **Employee** table in the **TEST** database as follows − −

```
mysql> use TEST;
mysql> create table Employees
   (
      id int not null,
      age int not null,
      first varchar (255),
      last varchar (255)
   );
Query OK, 0 rows affected (0.08 sec)
mysql>
```

Create Data Records

Let us now create a few records in the **Employee** table as follows − −

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)

mysql>
```

SELECT Operation

Following example shows how we can execute the **SQL SELECT** statement using JTSL in JSP programming –

```
<%@ page import = "java.io.*,java.util.*,java.sql.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
   <head>
      <title>SELECT Operation</title>
   </head>

   <body>
      <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
         url = "jdbc:mysql://localhost/TEST"
         user = "root"  password = "pass123"/>

      <sql:query dataSource = "${snapshot}" var = "result">
         SELECT * from Employees;
      </sql:query>

      <table border = "1" width = "100%">
         <tr>
            <th>Emp ID</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Age</th>
         </tr>

         <c:forEach var = "row" items = "${result.rows}">
            <tr>
               <td><c:out value = "${row.id}"/></td>
               <td><c:out value = "${row.first}"/></td>
               <td><c:out value = "${row.last}"/></td>
               <td><c:out value = "${row.age}"/></td>
            </tr>
         </c:forEach>
      </table>

   </body>
</html>
```

Access the above JSP, the following result will be displayed –

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30
103	Sumit	Mittal	28

INSERT Operation

Following example shows how we can execute the SQL INSERT statement using JTSL in JSP programming −

```
<%@ page import = "java.io.*,java.util.*,java.sql.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
   <head>
      <title>JINSERT Operation</title>
   </head>

   <body>
      <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
         url = "jdbc:mysql://localhost/TEST"
         user = "root"  password = "pass123"/>
         <sql:update dataSource = "${snapshot}" var = "result">
         INSERT INTO Employees VALUES (104, 2, 'Nuha', 'Ali');
      </sql:update>

      <sql:query dataSource = "${snapshot}" var = "result">
         SELECT * from Employees;
      </sql:query>

      <table border = "1" width = "100%">
         <tr>
            <th>Emp ID</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Age</th>
         </tr>

         <c:forEach var = "row" items = "${result.rows}">
            <tr>
               <td><c:out value = "${row.id}"/></td>
               <td><c:out value = "${row.first}"/></td>
               <td><c:out value = "${row.last}"/></td>
               <td><c:out value = "${row.age}"/></td>
            </tr>
         </c:forEach>
      </table>

   </body>
</html>
```

Access the above JSP, the following result will be displayed −

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30

103	Sumit	Mittal	28
104	Nuha	Ali	2

DELETE Operation

Following example shows how we can execute the **SQL DELETE** statement using JTSL in JSP programming −

```
<%@ page import = "java.io.*,java.util.*,java.sql.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
   <head>
      <title>DELETE Operation</title>
   </head>

   <body>
      <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
         url = "jdbc:mysql://localhost/TEST"
         user = "root" password = "pass123"/>

      <c:set var = "empId" value = "103"/>

      <sql:update dataSource = "${snapshot}" var = "count">
         DELETE FROM Employees WHERE Id = ?
         <sql:param value = "${empId}" />
      </sql:update>

      <sql:query dataSource = "${snapshot}" var = "result">
         SELECT * from Employees;
      </sql:query>

      <table border = "1" width = "100%">
         <tr>
            <th>Emp ID</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Age</th>
         </tr>

         <c:forEach var = "row" items = "${result.rows}">
            <tr>
               <td><c:out value = "${row.id}"/></td>
               <td><c:out value = "${row.first}"/></td>
               <td><c:out value = "${row.last}"/></td>
               <td><c:out value = "${row.age}"/></td>
            </tr>
         </c:forEach>
      </table>
   </body>
</html>
```

Access the above JSP, the following result will be displayed −

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30

UPDATE Operation

Following example shows how we can execute the **SQL UPDATE** statement using JTSL in JSP programming −

```
<%@ page import = "java.io.*,java.util.*,java.sql.*"%>
<%@ page import = "javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<%@ taglib uri = "http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
   <head>
      <title>DELETE Operation</title>
   </head>

   <body>
      <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
         url = "jdbc:mysql://localhost/TEST"
         user = "root" password = "pass123"/>

      <c:set var = "empId" value = "102"/>

      <sql:update dataSource = "${snapshot}" var = "count">
         UPDATE Employees SET WHERE last = 'Ali'
         <sql:param value = "${empId}" />
      </sql:update>

      <sql:query dataSource = "${snapshot}" var = "result">
         SELECT * from Employees;
      </sql:query>

      <table border = "1" width = "100%">
         <tr>
            <th>Emp ID</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Age</th>
         </tr>

         <c:forEach var = "row" items = "${result.rows}">
            <tr>
```

```
                <td><c:out value = "${row.id}"/></td>
                <td><c:out value = "${row.first}"/></td>
                <td><c:out value = "${row.last}"/></td>
                <td><c:out value = "${row.age}"/></td>
             </tr>
          </c:forEach>
       </table>

   </body>
</html>
```

Access the above JSP, the following result will be displayed –

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Ali	30

JSP - XML Data

When you send the XML data via HTTP, it makes sense to use JSP to handle incoming and outgoing XML documents; for example, RSS documents. As an XML document is merely a bunch of text, creating one through a JSP is much easier than creating an HTML document.

Sending XML from a JSP

You can send the XML content using JSPs the same way you send HTML. The only difference is that you must set the content type of your page to text/xml. To set the content type, use the **<%@page%>** tag, like this –

```
<%@ page contentType = "text/xml" %>
```

Following example will show how to send XML content to the browser –

```
<%@ page contentType = "text/xml" %>

<books>
   <book>
      <name>Padam History</name>
      <author>ZARA</author>
      <price>100</price>
   </book>
</books>
```

Access the above XML using different browsers to see the document tree presentation of the above XML.

Processing XML in JSP

Before you proceed with XML processing using JSP, you will need to copy the following two XML and XPath related libraries into your **<Tomcat Installation Directory>\lib** –

XercesImpl.jar – Download it from https://www.apache.org/dist/xerces/j/

xalan.jar – Download it from https://xml.apache.org/xalan-j/index.html

Let us put the following content in books.xml file –

```
<books>
   <book>
      <name>Padam History</name>
      <author>ZARA</author>
      <price>100</price>
   </book>

   <book>
      <name>Great Mistry</name>
      <author>NUHA</author>
      <price>2000</price>
   </book>
</books>
```

Try the following **main.jsp**, keeping in the same directory –

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

<html>
   <head>
      <title>JSTL x:parse Tags</title>
   </head>

   <body>
      <h3>Books Info:</h3>
      <c:import var = "bookInfo" url="http://localhost:8080/books.xml"/>

      <x:parse xml = "${bookInfo}" var = "output"/>
      <b>The title of the first book is</b>:
      <x:out select = "$output/books/book[1]/name" />
      <br>

      <b>The price of the second book</b>:
      <x:out select = "$output/books/book[2]/price" />
   </body>
</html>
```

Access the above JSP using **http://localhost:8080/main.jsp**, the following result will be displayed –

```
Books Info:
```

```
The title of the first book is:Padam History


The price of the second book: 2000
```

Formatting XML with JSP

Consider the following XSLT stylesheet **style.xsl** –

```
<?xml version = "1.0"?>
<xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
   version = "1.0">

   <xsl:output method = "html" indent = "yes"/>
   <xsl:template match = "/">
      <html>
         <body>
            <xsl:apply-templates/>
         </body>
      </html>
   </xsl:template>

   <xsl:template match = "books">
      <table border = "1" width = "100%">
         <xsl:for-each select = "book">
            <tr>
               <td>
                  <i><xsl:value-of select = "name"/></i>
               </td>

               <td>
                  <xsl:value-of select = "author"/>
               </td>

               <td>
                  <xsl:value-of select = "price"/>
               </td>
            </tr>
         </xsl:for-each>
      </table>

   </xsl:template>
</xsl:stylesheet>
```

Now consider the following JSP file –

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

<html>
   <head>
      <title>JSTL x:transform Tags</title>
   </head>

   <body>
      <h3>Books Info:</h3>
      <c:set var = "xmltext">
```

```
         <books>
            <book>
               <name>Padam History</name>
               <author>ZARA</author>
               <price>100</price>
            </book>

            <book>
               <name>Great Mistry</name>
               <author>NUHA</author>
               <price>2000</price>
            </book>
         </books>
      </c:set>

      <c:import url = "http://localhost:8080/style.xsl" var = "xslt"/>
      <x:transform xml = "${xmltext}" xslt = "${xslt}"/>
   </body>
</html>
```

The following result will be displayed −

```
Books Info:
```

Padam History	ZARA	100
Great Mistry	NUHA	2000

To know more about XML processing using JSTL, you can check JSP Standard Tag Library .

JSP - JavaBeans

A JavaBean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.

Following are the unique characteristics that distinguish a JavaBean from other Java classes −

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the **Serializable** interface.
- It may have a number of properties which can be read or written.
- It may have a number of "**getter**" and "**setter**" methods for the properties.

JavaBeans Properties

A JavaBean property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define.

A JavaBean property may be **read, write, read only**, or **write only**. JavaBean properties are accessed through two methods in the JavaBean's implementation class −

S.No.	Method & Description
1	get**PropertyName**() For example, if property name is *firstName*, your method name would be **getFirstName()** to read that property. This method is called accessor.
2	set**PropertyName**() For example, if property name is *firstName*, your method name would be **setFirstName()** to write that property. This method is called mutator.

A read-only attribute will have only a **getPropertyName()** method, and a write-only attribute will have only a **setPropertyName()** method.

JavaBeans Example

Consider a student class with few properties −

```
package com.tutorialspoint;

public class StudentsBean implements java.io.Serializable {
   private String firstName = null;
   private String lastName = null;
   private int age = 0;

   public StudentsBean() {
   }
   public String getFirstName(){
      return firstName;
   }
   public String getLastName(){
      return lastName;
   }
   public int getAge(){
      return age;
   }
   public void setFirstName(String firstName){
      this.firstName = firstName;
   }
   public void setLastName(String lastName){
      this.lastName = lastName;
   }
   public void setAge(Integer age){
      this.age = age;
   }
}
```

Accessing JavaBeans

The **useBean** action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows −

```
<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>
```

Here values for the scope attribute can be a **page, request, session** or **application based** on your requirement. The value of the **id** attribute may be any value as a long as it is a unique name among other **useBean declarations** in the same JSP.

Following example shows how to use the useBean action −

```
<html>
   <head>
      <title>useBean Example</title>
   </head>
   
   <body>
      <jsp:useBean id = "date" class = "java.util.Date" />
      <p>The date/time is <%= date %>
   </body>
</html>
```

You will receive the following result − −

```
The date/time is Thu Sep 30 11:18:11 GST 2010
```

Accessing JavaBeans Properties

Along with **<jsp:useBean...>** action, you can use the **<jsp:getProperty/>** action to access the get methods and the **<jsp:setProperty/>** action to access the set methods. Here is the full syntax −

```
<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">
   <jsp:setProperty name = "bean's id" property = "property name"
      value = "value"/>
   <jsp:getProperty name = "bean's id" property = "property name"/>
   ...........
</jsp:useBean>
```

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the **get** or the **set** methods that should be invoked.

Following example shows how to access the data using the above syntax −

```
<html>
   <head>
      <title>get and set properties Example</title>
   </head>

   <body>
      <jsp:useBean id = "students" class = "com.tutorialspoint.StudentsBean">
         <jsp:setProperty name = "students" property = "firstName" value = "Zara"/>
         <jsp:setProperty name = "students" property = "lastName" value = "Ali"/>
         <jsp:setProperty name = "students" property = "age" value = "10"/>
      </jsp:useBean>

      <p>Student First Name:
         <jsp:getProperty name = "students" property = "firstName"/>
      </p>

      <p>Student Last Name:
         <jsp:getProperty name = "students" property = "lastName"/>
      </p>

      <p>Student Age:
         <jsp:getProperty name = "students" property = "age"/>
      </p>

   </body>
</html>
```

Let us make the **StudentsBean.class** available in CLASSPATH. Access the above JSP. the following result will be displayed –

```
Student First Name: Zara


Student Last Name: Ali


Student Age: 10
```

JSP - Custom Tags

In this chapter, we will discuss the Custom Tags in JSP. A custom tag is a user-defined JSP language element. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The Web container then invokes those operations when the JSP page's servlet is executed.

JSP tag extensions lets you create new tags that you can insert directly into a JavaServer Page. The JSP 2.0 specification introduced the Simple Tag Handlers for writing these custom tags.

To write a custom tag, you can simply extend **SimpleTagSupport** class and override the **doTag()** method, where you can place your code to generate content for the tag.

Create "Hello" Tag

Consider you want to define a custom tag named <ex:Hello> and you want to use it in the following fashion without a body −

```
<ex:Hello />
```

To create a custom JSP tag, you must first create a Java class that acts as a tag handler. Let us now create the **HelloTag** class as follows −

```
package com.tutorialspoint;

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

public class HelloTag extends SimpleTagSupport {
   public void doTag() throws JspException, IOException {
      JspWriter out = getJspContext().getOut();
      out.println("Hello Custom Tag!");
   }
}
```

The above code has simple coding where the **doTag()** method takes the current JspContext object using the **getJspContext()** method and uses it to send **"Hello Custom Tag!"** to the current **JspWriter** object

Let us compile the above class and copy it in a directory available in the environment variable CLASSPATH. Finally, create the following tag library file: **<Tomcat-Installation-Directory>webapps\ROOT\WEB-INF\custom.tld**.

```
<taglib>
   <tlib-version>1.0</tlib-version>
   <jsp-version>2.0</jsp-version>
   <short-name>Example TLD</short-name>

   <tag>
      <name>Hello</name>
      <tag-class>com.tutorialspoint.HelloTag</tag-class>
      <body-content>empty</body-content>
   </tag>
</taglib>
```

Let us now use the above defined custom tag **Hello** in our JSP program as follows −

```
<%@ taglib prefix = "ex" uri = "WEB-INF/custom.tld"%>

<html>
   <head>
      <title>A sample custom tag</title>
   </head>

   <body>
      <ex:Hello/>
   </body>
</html>
```

Call the above JSP and this should produce the following result −

```
Hello Custom Tag!
```

Accessing the Tag Body

You can include a message in the body of the tag as you have seen with standard tags. Consider you want to define a custom tag named **<ex:Hello>** and you want to use it in the following fashion with a body −

```
<ex:Hello>
   This is message body
</ex:Hello>
```

Let us make the following changes in the above tag code to process the body of the tag −

```
package com.tutorialspoint;

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

public class HelloTag extends SimpleTagSupport {
   StringWriter sw = new StringWriter();
   public void doTag()

   throws JspException, IOException {
      getJspBody().invoke(sw);
      getJspContext().getOut().println(sw.toString());
   }
}
```

Here, the output resulting from the invocation is first captured into a **StringWriter** before being written to the JspWriter associated with the tag. We need to change TLD file as follows −

```
<taglib>
   <tlib-version>1.0</tlib-version>
   <jsp-version>2.0</jsp-version>
   <short-name>Example TLD with Body</short-name>

   <tag>
      <name>Hello</name>
      <tag-class>com.tutorialspoint.HelloTag</tag-class>
      <body-content>scriptless</body-content>
   </tag>
</taglib>
```

Let us now call the above tag with proper body as follows −

```
<%@ taglib prefix = "ex" uri = "WEB-INF/custom.tld"%>

<html>
   <head>
```

```
      <title>A sample custom tag</title>
   </head>

   <body>
      <ex:Hello>
         This is message body
      </ex:Hello>
   </body>
</html>
```

You will receive the following result –

```
This is message body
```

Custom Tag Attributes

You can use various attributes along with your custom tags. To accept an attribute value, a custom tag class needs to implement the **setter** methods, identical to the JavaBean setter methods as shown below –

```
package com.tutorialspoint;

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

public class HelloTag extends SimpleTagSupport {
   private String message;

   public void setMessage(String msg) {
      this.message = msg;
   }
   StringWriter sw = new StringWriter();
   public void doTag()

   throws JspException, IOException {
      if (message != null) {
         /* Use message from attribute */
         JspWriter out = getJspContext().getOut();
         out.println( message );
      } else {
         /* use message from the body */
         getJspBody().invoke(sw);
         getJspContext().getOut().println(sw.toString());
      }
   }
}
```

The attribute's name is **"message"**, so the setter method is **setMessage()**. Let us now add this attribute in the TLD file using the **<attribute>** element as follows –

```
<taglib>
   <tlib-version>1.0</tlib-version>
   <jsp-version>2.0</jsp-version>
   <short-name>Example TLD with Body</short-name>
   <tag>
      <name>Hello</name>
      <tag-class>com.tutorialspoint.HelloTag</tag-class>
      <body-content>scriptless</body-content>

      <attribute>
         <name>message</name>
      </attribute>

   </tag>
</taglib>
```

Let us follow JSP with message attribute as follows –

```
<%@ taglib prefix = "ex" uri = "WEB-INF/custom.tld"%>

<html>
   <head>
      <title>A sample custom tag</title>
   </head>

   <body>
      <ex:Hello message = "This is custom tag" />
   </body>
</html>
```

This will produce following result –

```
This is custom tag
```

Consider including the following properties for an attribute –

S.No.	Property & Purpose
1	**name** The name element defines the name of an attribute. Each attribute name must be unique for a particular tag.
2	**required** This specifies if this attribute is required or is an optional one. It would be false for optional.
3	**rtexprvalue** Declares if a runtime expression value for a tag attribute is valid
4	**type** Defines the Java class-type of this attribute. By default it is assumed as **String**

5	**description** Informational description can be provided.
6	**fragment** Declares if this attribute value should be treated as a **JspFragment**.

Following is the example to specify properties related to an attribute −

```
.....
   <attribute>
      <name>attribute_name</name>
      <required>false</required>
      <type>java.util.Date</type>
      <fragment>false</fragment>
   </attribute>
.....
```

If you are using two attributes, then you can modify your TLD as follows −

```
.....
   <attribute>
      <name>attribute_name1</name>
      <required>false</required>
      <type>java.util.Boolean</type>
      <fragment>false</fragment>
   </attribute>

   <attribute>
      <name>attribute_name2</name>
      <required>true</required>
      <type>java.util.Date</type>
   </attribute>
.....
```

JSP - Expression Language (EL)

JSP Expression Language (EL) makes it possible to easily access application data stored in JavaBeans components. JSP EL allows you to create expressions both **(a)** arithmetic and **(b)** logical. Within a JSP EL expression, you can use **integers, floating point numbers, strings, the built-in constants true and false** for boolean values, and null.

Simple Syntax

Typically, when you specify an attribute value in a JSP tag, you simply use a string. For example −

```
<jsp:setProperty name = "box" property = "perimeter" value = "100"/>
```

JSP EL allows you to specify an expression for any of these attribute values. A simple syntax for JSP EL is as follows −

```
${expr}
```

Here **expr** specifies the expression itself. The most common operators in JSP EL are **.** and **[]**. These two operators allow you to access various attributes of Java Beans and built-in JSP objects.

For example, the above syntax **<jsp:setProperty>** tag can be written with an expression like −

```
<jsp:setProperty name = "box" property = "perimeter"

   value = "${2*box.width+2*box.height}"/>
```

When the JSP compiler sees the **${}** form in an attribute, it generates code to evaluate the expression and substitues the value of expresson.

You can also use the JSP EL expressions within template text for a tag. For example, the **<jsp:text>** tag simply inserts its content within the body of a JSP. The following **<jsp:text>** declaration inserts **<h1>Hello JSP!</h1>** into the JSP output −

```
<jsp:text>
   <h1>Hello JSP!</h1>
</jsp:text>
```

You can now include a JSP EL expression in the body of a **<jsp:text>** tag (or any other tag) with the same **${}** syntax you use for attributes. For example −

```
<jsp:text>
   Box Perimeter is: ${2*box.width + 2*box.height}
</jsp:text>
```

EL expressions can use parentheses to group subexpressions. For example, **${(1 + 2) * 3} equals 9, but ${1 + (2 * 3)} equals 7**.

To deactivate the evaluation of EL expressions, we specify the **isELIgnored** attribute of the page directive as below −

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

The valid values of this attribute are true and false. If it is true, EL expressions are ignored when they appear in static text or tag attributes. If it is false, EL expressions are evaluated by the container.

Basic Operators in EL

JSP Expression Language (EL) supports most of the arithmetic and logical operators supported by Java. Following table lists out the most frequently used operators −

S.No.	Operator & Description
1	**.** Access a bean property or Map entry
2	**[]** Access an array or List element
3	**( )** Group a subexpression to change the evaluation order
4	**+** Addition
5	**-** Subtraction or negation of a value
6	***** Multiplication
7	**/ or div** Division
8	**% or mod** Modulo (remainder)
9	**== or eq** Test for equality
10	**!= or ne** Test for inequality
11	**< or lt** Test for less than
12	**> or gt**

	Test for greater than
13	**<= or le** Test for less than or equal
14	**>= or ge** Test for greater than or equal
15	**&& or and** Test for logical AND
16	**|| or or** Test for logical OR
17	**! or not** Unary Boolean complement
18	**empty** Test for empty variable values

Functions in JSP EL

JSP EL allows you to use functions in expressions as well. These functions must be defined in the custom tag libraries. A function usage has the following syntax −

```
${ns:func(param1, param2, ...)}
```

Where **ns** is the namespace of the function, **func** is the name of the function and **param1** is the first parameter value. For example, the function **fn:length**, which is part of the JSTL library. This function can be used as follows to get the length of a string.

```
${fn:length("Get my length")}
```

To use a function from any tag library (standard or custom), you must install that library on your server and must include the library in your JSP using the **<taglib>** directive as explained in the JSTL chapter.

JSP EL Implicit Objects

The JSP expression language supports the following implicit objects −

S.No	Implicit object & Description
1	**pageScope** Scoped variables from page scope
2	**requestScope** Scoped variables from request scope
3	**sessionScope** Scoped variables from session scope
4	**applicationScope** Scoped variables from application scope
5	**param** Request parameters as strings
6	**paramValues** Request parameters as collections of strings
7	**header** HTTP request headers as strings
8	**headerValues** HTTP request headers as collections of strings
9	**initParam** Context-initialization parameters
10	**cookie** Cookie values
11	**pageContext** The JSP PageContext object for the current page

You can use these objects in an expression as if they were variables. The examples that follow will help you understand the concepts −

The pageContext Object

The pageContext object gives you access to the pageContext JSP object. Through the pageContext object, you can access the request object. For example, to access the incoming query string for a request, you can use the following expression −

```
${pageContext.request.queryString}
```

The Scope Objects

The **pageScope, requestScope, sessionScope**, and **applicationScope** variables provide access to variables stored at each scope level.

For example, if you need to explicitly access the box variable in the application scope, you can access it through the applicationScope variable as **applicationScope.box**.

The param and paramValues Objects

The param and paramValues objects give you access to the parameter values normally available through the **request.getParameter** and **request.getParameterValues** methods.

For example, to access a parameter named order, use the expression **${param.order}** or **${param["order"]}**.

Following is the example to access a request parameter named username −

```
<%@ page import = "java.io.*,java.util.*" %>
<%String title = "Accessing Request Param";%>

<html>
   <head>
      <title><% out.print(title); %></title>
   </head>

   <body>
      <center>
         <h1><% out.print(title); %></h1>
      </center>

      <div align = "center">
         <p>${param["username"]}</p>
      </div>
   </body>
</html>
```

The param object returns single string values, whereas the paramValues object returns string arrays.

header and headerValues Objects

The header and headerValues objects give you access to the header values normally available through the **request.getHeader** and the **request.getHeaders** methods.

For example, to access a header named user-agent, use the expression **${header.user-agent}** or **${header["user-agent"]}**.

Following is the example to access a header parameter named user-agent –

```
<%@ page import = "java.io.*,java.util.*" %>
<%String title = "User Agent Example";%>

<html>
   <head>
      <title><% out.print(title); %></title>
   </head>

   <body>
      <center>
         <h1><% out.print(title); %></h1>
      </center>

      <div align = "center">
         <p>${header["user-agent"]}</p>
      </div>
   </body>
</html>
```

The output will somewhat be like the following –

```
User Agent Example

Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/4.0;
   SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729;
   Media Center PC 6.0; HPNTDF; .NET4.0C; InfoPath.2)
```

The header object returns single string values, whereas the headerValues object returns string arrays.

JSP - Exception Handling

In this chapter. we will discuss how to handle exceptions in JSP. When you are writing a JSP code, you might make coding errors which can occur at any part of the code. There may occur the following type of errors in your JSP code −

Checked exceptions

A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

Runtime exceptions

A runtime exception is an exception that probably could have been avoided by the programmer. As opposed to the checked exceptions, runtime exceptions are ignored at the time of compliation.

Errors

These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

We will further discuss ways to handle run time exception/error occuring in your JSP code.

Using Exception Object

The exception object is an instance of a subclass of Throwable (e.g., java.lang. NullPointerException) and is only available in error pages. Following table lists out the important methods available in the Throwable class.

S.No.	Methods & Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of **getMessage()**.
4	**public void printStackTrace()**

	Prints the result of **toString()** along with the stack trace to **System.err**, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

JSP gives you an option to specify **Error Page** for each JSP. Whenever the page throws an exception, the JSP container automatically invokes the error page.

Following is an example to specifiy an error page for a **main.jsp**. To set up an error page, use the **<%@ page errorPage = "xxx" %>** directive.

```
<%@ page errorPage = "ShowError.jsp" %>

<html>
   <head>
      <title>Error Handling Example</title>
   </head>

   <body>
      <%
         // Throw an exception to invoke the error page
         int x = 1;

         if (x == 1) {
            throw new RuntimeException("Error condition!!!");
         }
      %>
   </body>
</html>
```

We will now write one Error Handling JSP ShowError.jsp, which is given below. Notice that the error-handling page includes the directive **<%@ page isErrorPage = "true" %>**. This directive causes the JSP compiler to generate the exception instance variable.

```
<%@ page isErrorPage = "true" %>

<html>
   <head>
      <title>Show Error Page</title>
   </head>

   <body>
```

```
      <h1>Opps...</h1>
      <p>Sorry, an error occurred.</p>
      <p>Here is the exception stack trace: </p>
      <pre><% exception.printStackTrace(response.getWriter()); %></pre>
   </body>
</html>
```

Access the **main.jsp**, you will receive an output somewhat like the following −

```
java.lang.RuntimeException: Error condition!!!
......

Opps...
Sorry, an error occurred.

Here is the exception stack trace:
```

Using JSTL Tags for Error Page

You can make use of JSTL tags to write an error page **ShowError.jsp**. This page has almost same logic as in the above example, with better structure and more information −

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

<html>
   <head>
      <title>Show Error Page</title>
   </head>

   <body>
      <h1>Opps...</h1>
      <table width = "100%" border = "1">
         <tr valign = "top">
            <td width = "40%"><b>Error:</b></td>
            <td>${pageContext.exception}</td>
         </tr>

         <tr valign = "top">
            <td><b>URI:</b></td>
            <td>${pageContext.errorData.requestURI}</td>
         </tr>

         <tr valign = "top">
            <td><b>Status code:</b></td>
            <td>${pageContext.errorData.statusCode}</td>
         </tr>

         <tr valign = "top">
            <td><b>Stack trace:</b></td>
            <td>
               <c:forEach var = "trace"
                  items = "${pageContext.exception.stackTrace}">
                  <p>${trace}</p>
               </c:forEach>
            </td>
         </tr>
      </table>

   </body>
</html>
```

Access the main.jsp, the following will be generated −

```
                              Opps...

Error:          java.lang.RuntimeException: Error condition!!!

URI:            /main.jsp

Status code:    500

                org.apache.jsp.main_jsp._jspService(main_jsp.java:65)

                org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:68)

Stack trace:    javax.servlet.http.HttpServlet.service(HttpServlet.java:722)

                org.apache.jasper.servlet.JspServlet.service(JspServlet.java:265)

                javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
```

Using Try...Catch Block

If you want to handle errors within the same page and want to take some action instead of firing an error page, you can make use of the **try....catch** block.

Following is a simple example which shows how to use the try...catch block. Let us put the following code in main.jsp −

```
<html>
   <head>
      <title>Try...Catch Example</title>
   </head>
   <body>
      <%
         try {
            int i = 1;
            i = i / 0;
            out.println("The answer is " + i);
         }
         catch (Exception e) {
            out.println("An exception occurred: " + e.getMessage());
         }
      %>
   </body>
</html>
```

Access the main.jsp, it should generate an output somewhat like the following −

```
An exception occurred: / by zero
```

JSP - Debugging

In this chapter, we will discuss Debugging a JSP. It is always difficult testing/debugging a JSP and servlets. JSP and Servlets tend to involve a large amount of client/server interaction, making errors likely but hard to reproduce.

The following are a few hints and suggestions that may aid you in your debugging.

Using System.out.println()

System.out.println() is easy to use as a marker to test whether a certain piece of code is being executed or not. We can print out variable values as well. Consider the following additional points −

> Since the System object is part of the core Java objects, it can be used everywhere without the need to install any extra classes. This includes **Servlets, JSP, RMI, EJB's, ordinary Beans** and **classes**, and **standalone applications**.
>
> Compared to stopping at breakpoints, writing to **System.out** doesn't interfere much with the normal execution flow of the application, which makes it very valuable when the iming is crucial.

Following is the syntax to use **System.out.println()** −

```
System.out.println("Debugging message");
```

Following example shows how to use **System.out.print()** −

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

<html>
   <head><title>System.out.println</title></head>
   <body>
      <c:forEach var = "counter" begin = "1" end = "10" step = "1" >
```

```
         <c:out value = "${counter-5}"/></br>
         <% System.out.println( "counter = " + pageContext.findAttribute("counter") ); %>
      </c:forEach>

   </body>
</html>
```

Access the above JSP, the browser will show the following result –

```
-4

-3

-2

-1

0

1

2

3

4

5
```

If you are using Tomcat, you will also find these lines appended to the end of **stdout.log** in the logs directory.

```
counter = 1

counter = 2

counter = 3

counter = 4

counter = 5

counter = 6

counter = 7

counter = 8

counter = 9

counter = 10
```

This way you can bring variables and other information into the system log which can be analyzed to find out the root cause of the problem or for various other reasons.

Using the JDB Logger

The **J2SE** logging framework is designed to provide logging services for any class running in the JVM. We can make use of this framework to log any information.

Let us re-write the above example using JDK logger API –

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

<html>
   <head><title>Logger.info</title></head>
   <body>
      <% Logger logger = Logger.getLogger(this.getClass().getName());%>

      <c:forEach var = "counter" begin = "1" end = "10" step = "1" >
      <c:set var = "myCount" value = "${counter-5}" />
      <c:out value = "${myCount}"/></br>
         <% String message = "counter = "
            + pageContext.findAttribute("counter") + "myCount = "
            + pageContext.findAttribute("myCount");
            logger.info( message );
         %>
      </c:forEach>

   </body>
</html>
```

The above code will generate similar result on the browser and in stdout.log, but you will have additional information in **stdout.log**. We will use the **info** method of the logger because and log the message just for informational purpose. Following is a snapshot of the stdout.log file –

```
24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService

INFO: counter = 1 myCount = -4

24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService

INFO: counter = 2 myCount = -3

24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService

INFO: counter = 3 myCount = -2

24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService

INFO: counter = 4 myCount = -1

24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService

INFO: counter = 5 myCount = 0

24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService

INFO: counter = 6 myCount = 1

24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService

INFO: counter = 7 myCount = 2

24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService

INFO: counter = 8 myCount = 3

24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService

INFO: counter = 9 myCount = 4

24-Sep-2010 23:31:31 org.apache.jsp.main_jsp _jspService

INFO: counter = 10 myCount = 5
```

Messages can be sent at various levels by using the convenience functions **severe(), warning(), info(), config(), fine(), finer(),** and **finest()**. Here finest() method can be used to log finest information and the severe() method can be used to log severe information.

You can use Log4J Framework to log messages in different files based on their severity levels and importance.

Debugging Tools

NetBeans is a free and open-source Java Integrated Development Environment that supports the development of standalone Java applications and Web applications supporting the JSP and servlet specifications and includes a JSP debugger as well.

NetBeans supports the following basic debugging functionalities −

- Breakpoints
- Stepping through code
- Watchpoints

You can refere to **NetBeans documentation** to understand above debugging functionalities.

Using JDB Debugger

You can debug JSP and servlets with the same **jdb** commands you use to debug an applet or an application.

To debug a JSP or servlet, you can debug **sun.servlet.http.HttpServer**, then observe as HttpServer executes the JSP/servlets in response to HTTP requests we make from a browser. This is very similar to how applets are debugged. The difference is that with applets, the actual program being debugged is **sun.applet.AppletViewer**.

Most debuggers hide this detail by automatically knowing how to debug applets. Until they do the same for JSP, you have to help your debugger by considering the following −

- Set your debugger's classpath. This helps you find **sun.servlet.http.Http-Server** and the associated classes.
- Set your debugger's classpath. This helps you find your JSP and support classes, typically **ROOT\WEB-INF\classes**.

Once you have set the proper classpath, start debugging **sun.servlet.http.HttpServer**. You can set breakpoints in whatever JSP you're interested in debugging, then use a web browser to make a request to the HttpServer for the given JSP **(http://localhost:8080/JSPToDebug)**. The execution here stops at breakpoints.

Using Comments

Comments in your code can help the debugging process in various ways. Comments can be used in lots of other ways in the debugging process.

The JSP uses Java comments and **single line (// ...)** and **multiple line (/* ... */)** comments can be used to temporarily remove parts of your Java code. If the bug disappears, take a closer look at the code you just commented and find out the problem.

Client and Server Headers

Sometimes when a JSP doesn't behave as expected, it's useful to look at the raw HTTP request and response. If you're familiar with the structure of HTTP, you can read the request and response and see what exactly is going with those headers.

Important Debugging Tips

Here is a list of some more debugging tips on JSP debugging −

- Ask a browser to show the raw content of the page it is displaying. This can help identify formatting problems. It's usually an option under the View menu.

- Make sure the browser isn't caching a previous request's output by forcing a full reload of the page. With **Netscape Navigator**, use **Shift-Reload**; with **Internet Explorer** use **Shift-Refresh**.

JSP - Security

JavaServer Pages and servlets make several mechanisms available to Web developers to secure applications. Resources are protected declaratively by identifying them in the application deployment descriptor and assigning a role to them.

Several levels of authentication are available, ranging from basic authentication using identifiers and passwords to sophisticated authentication using certificates.

Role Based Authentication

The authentication mechanism in the servlet specification uses a technique called **role-based security**. The idea is that rather than restricting resources at the user level, you create roles and restrict the resources by role.

You can define different roles in file **tomcat-users.xml**, which is located off Tomcat's home directory in conf. An example of this file is shown below −

```
<?xml version = '1.0' encoding = 'utf-8'?>
<tomcat-users>
   <role rolename = "tomcat"/>
   <role rolename = "role1"/>
   <role rolename = "manager"/>
   <role rolename = "admin"/>
   <user username = "tomcat" password = "tomcat" roles = "tomcat"/>
   <user username = "role1" password = "tomcat" roles = "role1"/>
   <user username = "both" password = "tomcat" roles = "tomcat,role1"/>
   <user username = "admin" password = "secret" roles = "admin,manager"/>
</tomcat-users>
```

This file defines a simple mapping between **username, password**, and **role**. Notice that a given user may have multiple roles; for example, **username = "both"** is in the "tomcat" role and the "role1" role.

Once you have identified and defined different roles, role-based security restrictions can be placed on different Web Application resources by using the **<security-constraint>** element in **web.xml** file available in the WEB-INF directory.

Following is a sample entry in web.xml −

```
<web-app>
   ...
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>SecuredBookSite</web-resource-name>
         <url-pattern>/secured/*</url-pattern>
         <http-method>GET</http-method>
         <http-method>POST</http-method>
      </web-resource-collection>

      <auth-constraint>
         <description>
            Let only managers use this app
         </description>
         <role-name>manager</role-name>
      </auth-constraint>
   </security-constraint>

   <security-role>
      <role-name>manager</role-name>
   </security-role>

   <login-config>
      <auth-method>BASIC</auth-method>
   </login-config>
   ...
</web-app>
```

The above entries would mean −

Any HTTP GET or POST request to a URL matched by /secured/* would be subject to the security restriction.

A person with the role of a manager is given access to the secured resources.

The **login-config** element is used to describe the **BASIC** form of authentication.

If you try browsing any URL including the **/security** directory, the following dialog box will be displayed asking for username and password. If you provide a user **"admin"** and password **"secrer"**, then you will have access on the URL matched by **/secured/*** as we have defined the user admin with manager role who is allowed to access this resource.

Form Based Authentication

When you use the FORM authentication method, you must supply a login form to prompt the user for a username and password. Following is a simple code of **login.jsp**. This helps create a form for the same purpose −

```
<html>
   <body bgcolor = "#ffffff">
      
      <form method = "POST" action ="j_security_check">
         <table border = "0">
            <tr>
               <td>Login</td>
               <td><input type = "text" name="j_username"></td>
            </tr>
            <tr>
               <td>Password</td>
               <td><input type = "password" name="j_password"></td>
            </tr>
         </table>
         <input type = "submit" value = "Login!">
      </form>
      
   </body>
</html>
```

Here you have to make sure that the login form must contain the form elements named **j_username** and **j_password**. The action in the **<form>** tag must be **j_security_check**. **POST** must be used as the form method. At the same time, you will have to modify the **<login-config>** tag to specify the auth-method as FORM −

```
<web-app>
   ...
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>SecuredBookSite</web-resource-name>
         <url-pattern>/secured/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
      </web-resource-collection>
      
      <auth-constraint>
         <description>Let only managers use this app</description>
         <role-name>manager</role-name>
      </auth-constraint>
   </security-constraint>
   
   <security-role>
      <role-name>manager</role-name>
   </security-role>
   
   <login-config>
```

```
      <auth-method>FORM</auth-method>
      <form-login-config>
         <form-login-page>/login.jsp</form-login-page>
         <form-error-page>/error.jsp</form-error-page>
      </form-login-config>
   </login-config>
   ...
</web-app>
```

Now when you try to access any resource with **URL /secured/***, it will display the above form asking for the user id and password. When the container sees the "**j_security_check**" action, it uses some internal mechanism to authenticate the caller.

If the login succeeds and the caller is authorized to access the secured resource, then the container uses a session-id to identify a login session for the caller from that point on. The container maintains the login session with a cookie containing the session-id. The server sends the cookie back to the client, and as long as the caller presents this cookie with subsequent requests, then the container will know who the caller is.

If the login fails, then the server sends back the page identified by the form-error-page setting

Here, **j_security_check** is the action that applications using form based login have to specify for the login form. In the same form, you should also have a text input control called **j_username** and a **password input control** called **j_password**. When you see this, it means that the information contained in the form will be submitted to the server, which will check name and password. How this is done is server specific.

Check Standard Realm Implementations to understand how **j_security_check** works for Tomcat container..

Programmatic Security in a Servlet/JSP

The **HttpServletRequest** object provides the following methods, which can be used to mine security information at runtime −

S.No.	Method & Description
1	**String getAuthType()** The **getAuthType()** method returns a String object that represents the name of the authentication scheme used to protect the Servlet.
2	**boolean isUserInRole(java.lang.String role)** The **isUserInRole()** method returns a boolean value: true if the user is in the given role or false if they are not.

3	**String getProtocol()** The **getProtocol()** method returns a String object representing the protocol that was used to send the request. This value can be checked to determine if a secure protocol was used.
4	**boolean isSecure()** The **isSecure()** method returns a boolean value representing if the request was made using HTTPS. A value of true means it was and the connection is secure. A value of false means the request was not.
5	**Principle getUserPrinciple()** The **getUserPrinciple()** method returns a java.security.Principle object that contains the name of the current authenticated user.

For example, for a JavaServer Page that links to pages for managers, you might have the following code –

```
<% if (request.isUserInRole("manager")) { %>
   <a href = "managers/mgrreport.jsp">Manager Report</a>
   <a href = "managers/personnel.jsp">Personnel Records</a>
<% } %>
```

By checking the user's role in a JSP or servlet, you can customize the Webpage to show the user only the items she can access. If you need the user's name as it was entered in the authentication form, you can call the **getRemoteUser** method in the request object.

JSP - Internationalization| i18n| l10n

In this chapter, we will discuss the concept of Internationalization in JSP. Before we proceed, let us understand the following three important terms –

- **Internationalization (i18n)** – This means enabling a website to provide different versions of content translated into the visitor's language or nationality.
- **Localization (l10n)** – This means adding resources to a website to adapt it to a particular geographical or cultural region for example Hindi translation to a web site.
- **locale** – This is a particular cultural or geographical region. It is usually referred to as a language symbol followed by a country symbol which are separated by an underscore. For example, "**en_US**" represents english locale for US.

There are a number of items which should be taken care of while building up a global Website. This tutorial will not give you complete detail on this but it will give you a good example on how you can offer your Webpage in different languages to the internet community by differentiating their location, i.e., locale.

A JSP can pick up appropriate version of the site based on the requester's locale and provide appropriate site version according to the local language, culture and requirements. Following is the method of request object which returns the Locale object.

```
java.util.Locale request.getLocale()
```

Detecting Locale

Following are the important locale methods which you can use to detect **requester's location, language** and of course **locale**. All the below methods display the country name and language name set in the requester's browser.

S.No.	Method & Description
1	**String getCountry()** This method returns the country/region code in upper case for this locale in ISO 3166 2-letter format.
2	**String getDisplayCountry()** This method returns a name for the locale's country that is appropriate for display to the user.
3	**String getLanguage()** This method returns the language code in lower case for this locale in ISO 639 format.
4	**String getDisplayLanguage()** This method returns a name for the locale's language that is appropriate for display to the user.
5	**String getISO3Country()** This method returns a three-letter abbreviation for this locale's country.
6	**String getISO3Language()** This method returns a three-letter abbreviation for this locale's language.

Example

The following example shows how to display a language and associated country for a request in a JSP −

```
<%@ page import = "java.io.*,java.util.Locale" %>
<%@ page import = "javax.servlet.*,javax.servlet.http.* "%>
<%
   //Get the client's Locale
   Locale locale = request.getLocale();
   String language = locale.getLanguage();
   String country = locale.getCountry();
%>

<html>
   <head>
      <title>Detecting Locale</title>
   </head>

   <body>
      <center>
         <h1>Detecting Locale</h1>
      </center>

      <p align = "center">
         <%
            out.println("Language : " + language  + "<br />");
            out.println("Country  : " + country   + "<br />");
         %>
      </p>
   </body>
</html>
```

Languages Setting

A JSP can output a page written in a Western European language such as English, Spanish, German, French, Italian, Dutch etc. Here it is important to set Content-Language header to display all the characters properly.

Another important point is to display all the special characters using HTML entities; for example, **"&#241;"** represents **"ñ"**, and **"&#161;"** represents **"¡"** as follows −

```
<%@ page import = "java.io.*,java.util.Locale" %>
<%@ page import = "javax.servlet.*,javax.servlet.http.* "%>

<%
   // Set response content type
   response.setContentType("text/html");

   // Set spanish language code.
   response.setHeader("Content-Language", "es");
   String title = "En Español";
%>
```

```
<html>
   <head>
      <title><%  out.print(title); %></title>
   </head>

   <body>
      <center>
         <h1><%  out.print(title); %></h1>
      </center>

      <div align = "center">
         <p>En Español</p>
         <p>¡Hola Mundo!</p>
      </div>
   </body>
</html>
```

Locale Specific Dates

You can use the **java.text.DateFormat** class and its static **getDateTimeInstance( )** method to format date and time specific to locale. Following is the example which shows how to format dates specific to a given locale −

```
<%@ page import = "java.io.*,java.util.Locale" %>
<%@ page import = "javax.servlet.*,javax.servlet.http.* "%>
<%@ page import = "java.text.DateFormat,java.util.Date" %>

<%
   String title = "Locale Specific Dates";

   //Get the client's Locale
   Locale locale = request.getLocale( );

   String date = DateFormat.getDateTimeInstance(
      DateFormat.FULL,
      DateFormat.SHORT,
      locale).format(new Date( ));
%>

<html>

   <head>
      <title><% out.print(title); %></title>
   </head>

   <body>
      <center>
         <h1><% out.print(title); %></h1>
      </center>

      <div align = "center">
         <p>Local Date: <%  out.print(date); %></p>
      </div>
   </body>
</html>
```

Locale Specific Currency

You can use the **java.txt.NumberFormat** class and its static **getCurrencyInstance( )** method to format a number, such as a long or double type, in a locale specific curreny. Following is the example which shows how to format currency specific to a given locale –

```
<%@ page import = "java.io.*,java.util.Locale" %>
<%@ page import = "javax.servlet.*,javax.servlet.http.* "%>
<%@ page import = "java.text.NumberFormat,java.util.Date" %>

<%
   String title = "Locale Specific Currency";

   //Get the client's Locale
   Locale locale = request.getLocale( );

   NumberFormat nft = NumberFormat.getCurrencyInstance(locale);
   String formattedCurr = nft.format(1000000);
%>

<html>

   <head>
      <title><% out.print(title); %></title>
   </head>

   <body>
      <center>
         <h1><% out.print(title); %></h1>
      </center>

      <div align = "center">
         <p>Formatted Currency: <%  out.print(formattedCurr); %></p>
      </div>
   </body>
</html>
```

Locale Specific Percentage

You can use the **java.txt.NumberFormat** class and its static **getPercentInstance( )** method to get locale specific percentage. Following example shows how to format percentage specific to a given locale –

```
<%@ page import = "java.io.*,java.util.Locale" %>
<%@ page import = "javax.servlet.*,javax.servlet.http.* "%>
<%@ page import = "java.text.NumberFormat,java.util.Date" %>

<%
   String title = "Locale Specific Percentage";

   //Get the client's Locale
   Locale locale = request.getLocale( );

   NumberFormat nft = NumberFormat.getPercentInstance(locale);
   String formattedPerc = nft.format(0.51);
```

```
%>

<html>

   <head>
      <title><% out.print(title); %></title>
   </head>

   <body>
      <center>
         <h1><% out.print(title); %></h1>
      </center>

      <div align = "center">
         <p>Formatted Percentage: <%  out.print(formattedPerc); %></p>
      </div>
   </body>
</html>
```

Previous Page   Next Page