JSP

1. 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 is the list of 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.

2. 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 http://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:

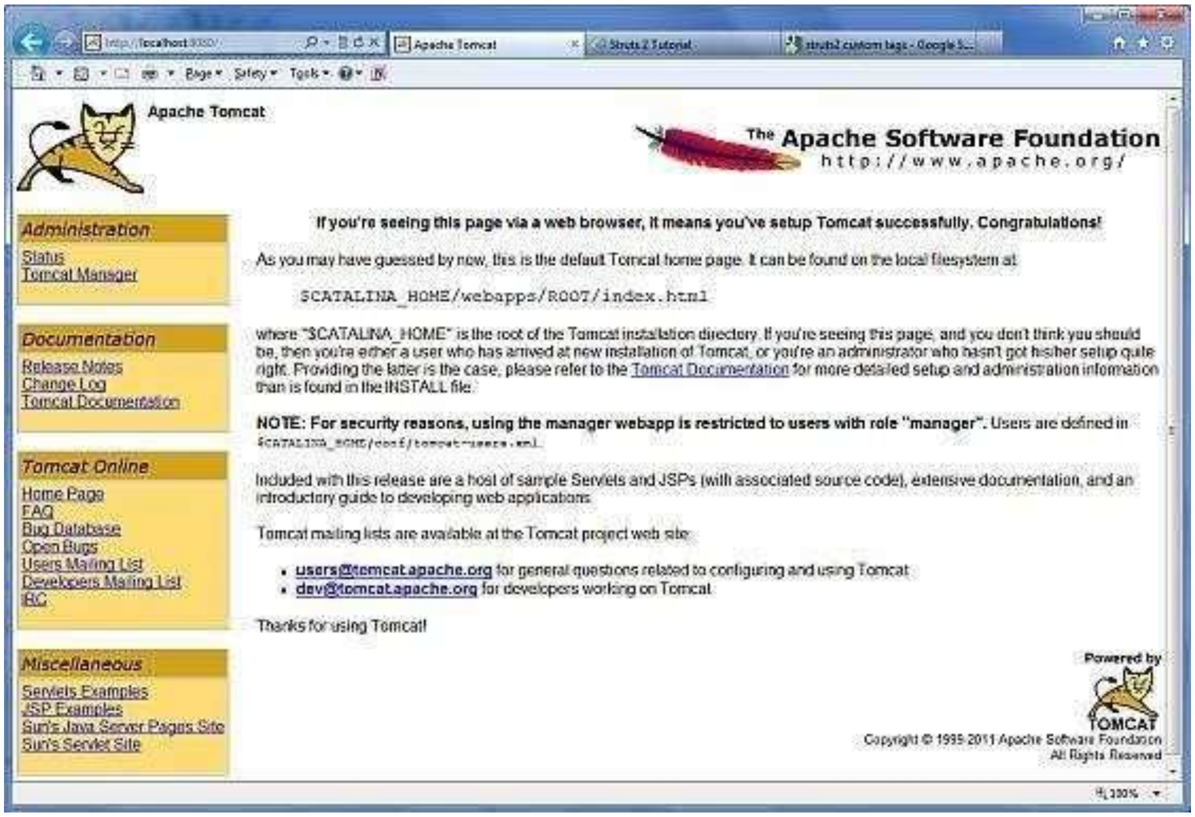
```
$CATALINA_HOME/bin/startup.sh


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/

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!

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

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: **http://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
```

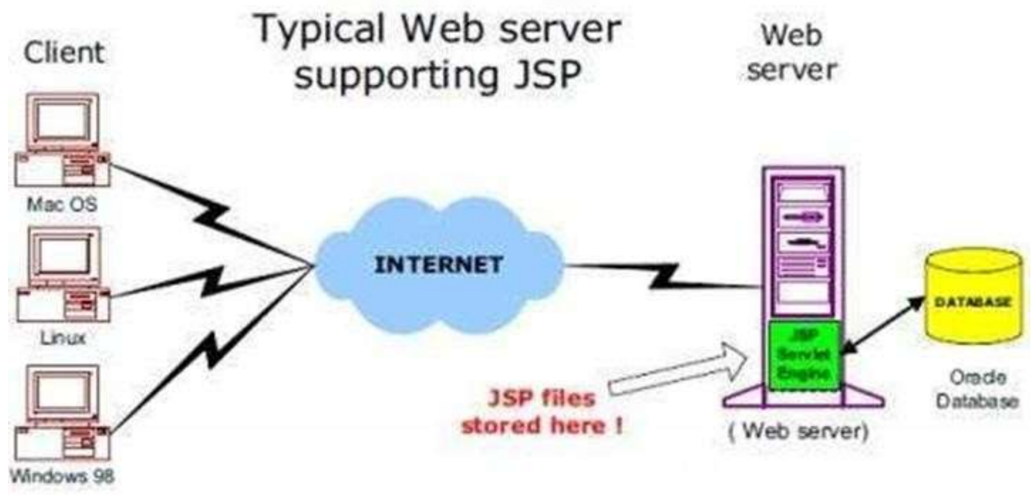
NOTE: Assuming that your development directory is **C:\JSPDev (Windows)** or **/usr/JSPDev (Unix)**, then you would need to add these directories as well in CLASSPATH.

3. 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.



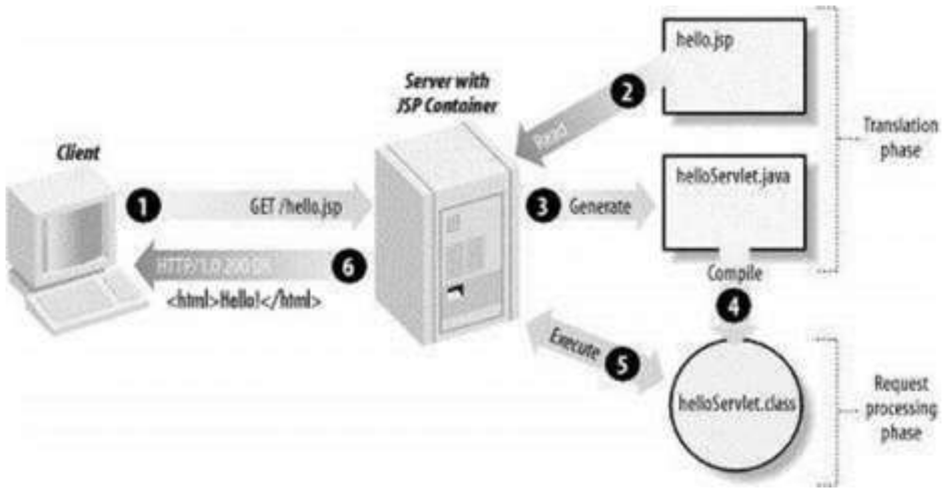
JSP Processing

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. This output is further 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

4. 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.

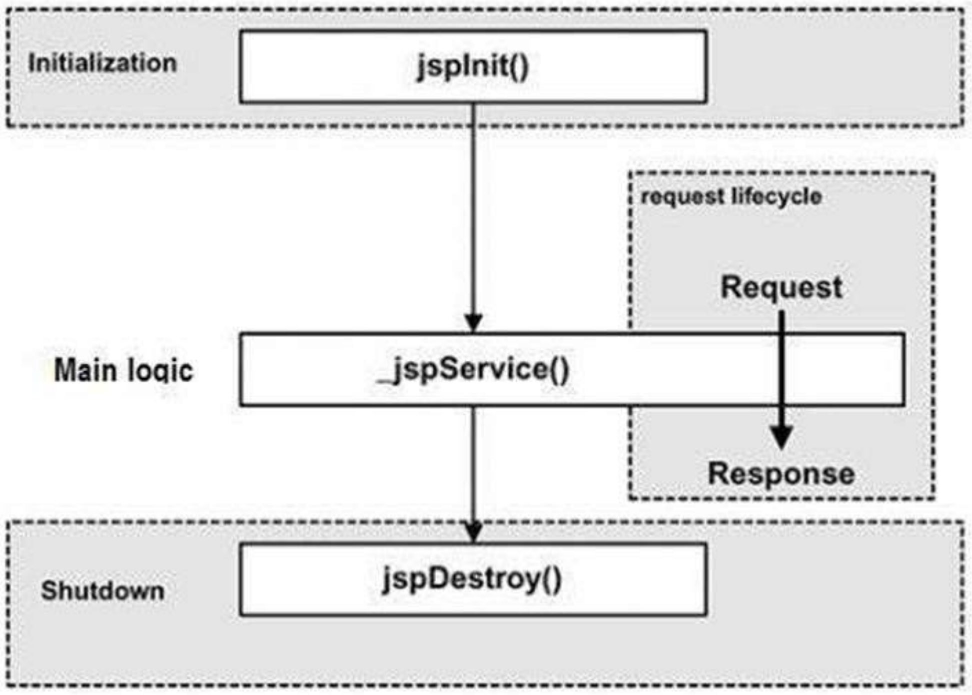
A JSP life cycle is defined as the process from its creation till the destruction. This is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet.

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.
}
```

5. 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:

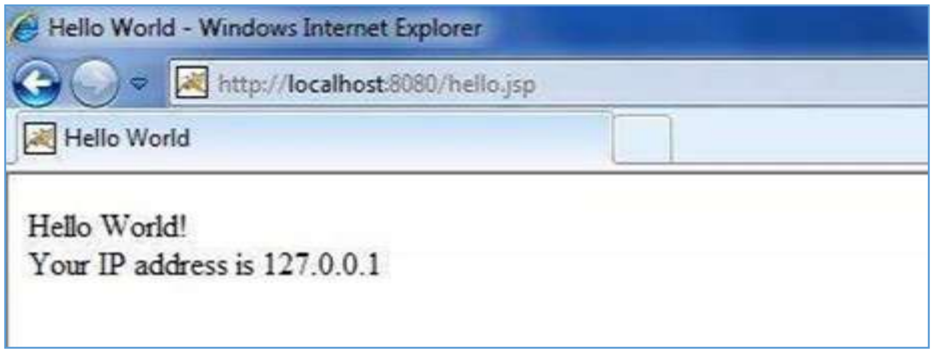
```
<jsp:scriptlet>
   code fragment
</jsp:scriptlet>
```

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-tomcat-7.0.2\webapps\ROOT** directory. Browse through the same using URL **http://localhost:8080/hello.jsp**. This would 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 %>
```

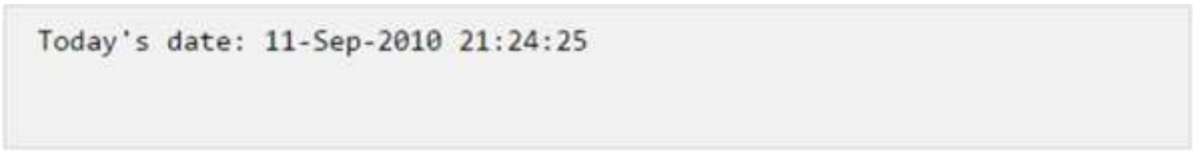
You can write the XML equivalent of the above syntax as follows:

```
<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 the text or the 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.

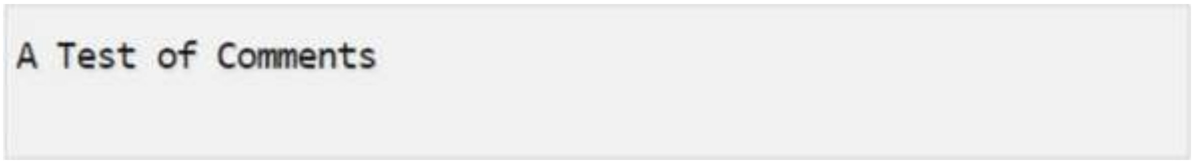
Following is the syntax of the JSP comments:

```
<%-- 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:

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

JSP Directives

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

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

There are three types of directive tag:

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

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:

Syntax	Purpose
jsp:include	Includes a file at the time the page is requested.
jsp:useBean	Finds or instantiates a JavaBean.

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

JSP Implicit Objects

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

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

Control-Flow Statements

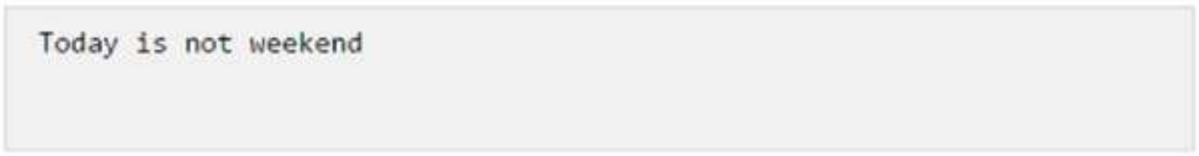
You can use all the APIs and building blocks of Java in your JSP programming including decision-making statements, loops, etc.

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:

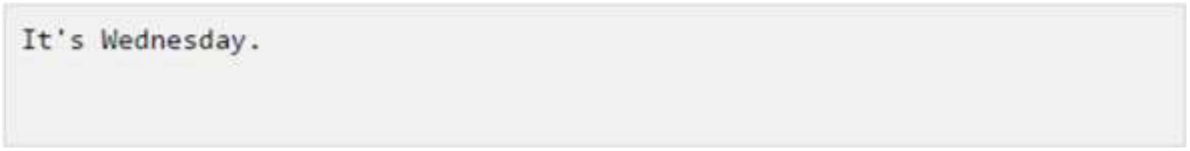


```
Today is not weekend
```

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.

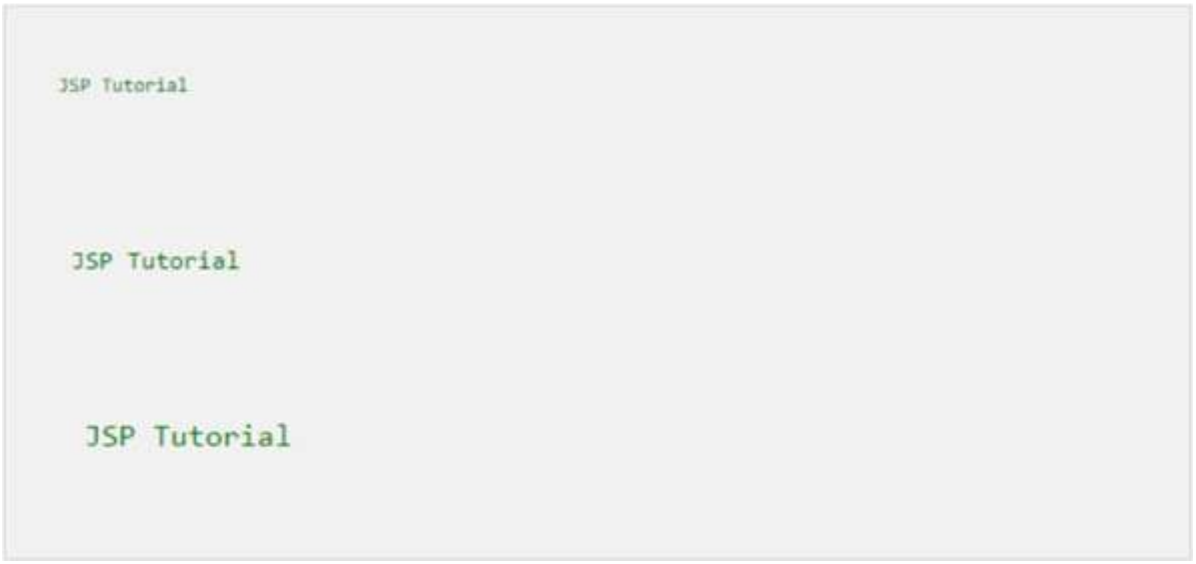
Let us look at the following **for** loop example:

```
<%! 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:

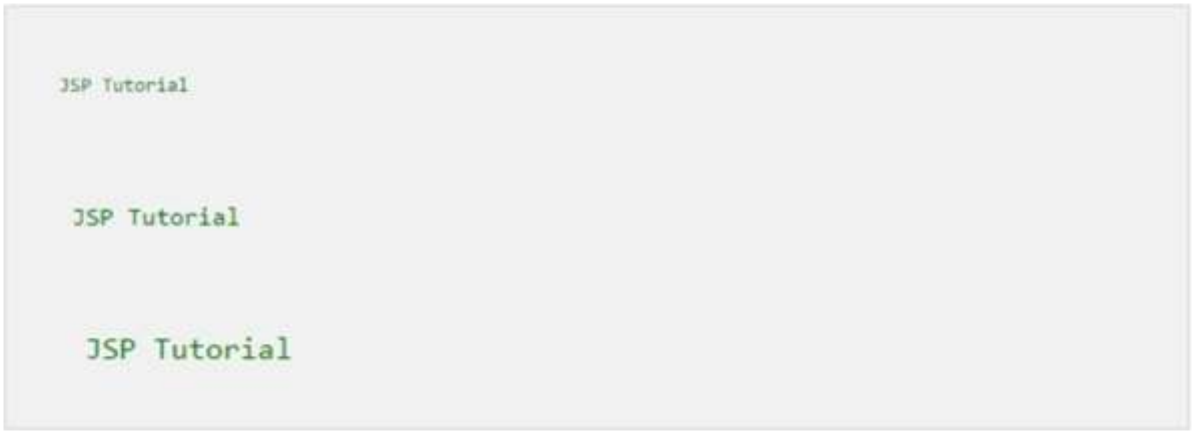


```
JSP Tutorial
JSP Tutorial
JSP Tutorial
```

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

6. 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:

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

The page Directive

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

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

isELIgnored	Specifies whether or not the EL expression within the JSP page will be ignored.
isScriptingEnabled	Determines if the scripting elements are allowed for use.

JSP - The page Directive

The **page** directive is used to provide instructions to the container that pertain to the current JSP page. You may code the page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.

Following is the basic syntax of page directive:

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

You can write 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:

Attribute	Purpose
buffer	Specifies a buffering model for the output stream.
autoFlush	Controls the behavior of the servlet output buffer.
contentType	Defines the character encoding scheme.
errorPage	Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
isErrorPage	Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.

extends	Specifies a superclass that the generated servlet must extend.
import	Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
info	Defines a string that can be accessed with the servlet's getServletInfo() method.
isThreadSafe	Defines the threading model for the generated servlet.
language	Defines the programming language used in the JSP page.
session	Specifies whether or not the JSP page participates in HTTP sessions.
isELIgnored	Specifies whether or not the EL expression within the JSP page will be ignored.
isScriptingEnabled	Determines if the scripting elements are allowed for use.

The buffer Attribute

The **buffer** attribute specifies the buffering characteristics for the server output response object.

You may code a value of "**none**" to specify no buffering so that the servlet output is immediately directed to the response object or you may code a maximum buffer size in kilobytes, which directs the servlet to write to the buffer before writing to the response object.

To direct the servlet to write the output directly to the **response output object**, use the following:

```
<%@ page buffer="none" %>
```

Use the following to direct the servlet to write the output to a buffer of size not less than 8 kilobytes:

```
<%@ page buffer="8kb" %>
```

The autoFlush Attribute

The autoFlush attribute specifies whether buffered output should be flushed automatically when the buffer is filled, or whether an exception should be raised to indicate the buffer overflow.

A value of **true (default)** indicates automatic buffer flushing and a value of false throws an exception.

The following directive causes the servlet to throw an exception when the servlet's output buffer is full:

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

This directive causes the servlet to flush the output buffer when full:

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

Usually, the buffer and the autoFlush attributes are coded on a single page directive as follows:

```
<%@ page buffer="16kb" autoflush="true" %>
```

The contentType Attribute

The contentType attribute sets the character encoding for the JSP page and for the generated response page. The default content type is **text/html**, which is the standard content type for HTML pages.

If you want to write out XML from your JSP, use the following page directive:

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

The following statement directs the browser to render the generated page as HTML:

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

The following directive sets the content type as a Microsoft Word document:

```
<%@ page contentType="application/msword" %>
```

You can also specify the character encoding for the response. For example, if you wanted to specify that the resulting page that is returned to the browser uses **ISO Latin 1**, you can use the following page directive:

```
<%@ page contentType="text/html:charset=ISO-8859-1" %>
```

The errorPage Attribute

The **errorPage** attribute tells the JSP engine which page to display if there is an error while the current page runs. The value of the errorPage attribute is a relative URL.

The following directive displays MyErrorPage.jsp when all uncaught exceptions are thrown:

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

The isErrorPage Attribute

The **isErrorPage** attribute indicates that the current JSP can be used as the error page for another JSP.

The value of isErrorPage is either true or false. The default value of the isErrorPage attribute is false.

For example, the **handleError.jsp** sets the isErrorPage option to true because it is supposed to handle errors:

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

The extends Attribute

The **extends** attribute specifies a superclass that the generated servlet must extend.

For example, the following directive directs the JSP translator to generate the servlet such that the servlet extends ***somePackage.SomeClass***:

```
<%@ page extends="somePackage.SomeClass" %>
```

The import Attribute

The **import** attribute serves the same function as, and behaves like, the Java import statement. The value for the import option is the name of the package you want to import.

To import **java.sql.***, use the following page directive:

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

To import multiple packages, you can specify them separated by comma as follows:

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

By default, a container automatically imports **java.lang.***, **javax.servlet.***, **javax.servlet.jsp.***, and **javax.servlet.http.***.

The info Attribute

The **info** attribute lets you provide a description of the JSP. The following is a coding example:

```
<%@ page info="This JSP Page Written By ZARA"  %>
```

The isThreadSafe Attribute

The **isThreadSafe** option marks a page as being thread-safe. By default, all JSPs are considered thread-safe. If you set the isThreadSafe option to false, the JSP engine makes sure that only one thread at a time is executing your JSP.

The following page directive sets the **isThreadSafe** option to false:

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

The language Attribute

The **language** attribute indicates the programming language used in scripting the JSP page.

For example, because you usually use Java as the scripting language, your language option looks like this:

```
<%@ page language="java" %>
```

The session Attribute

The **session** attribute indicates whether or not the JSP page uses HTTP sessions. A value of true means that the JSP page has access to a builtin **session** object and a value of false means that the JSP page cannot access the builtin session object.

Following directive allows the JSP page to use any of the builtin object session methods such as **session.getCreationTime()** or **session.getLastAccessTime()**:

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

The isELIgnored Attribute

The **isELIgnored** attribute gives you the ability to disable the evaluation of Expression Language (EL) expressions which has been introduced in JSP 2.0.

The default value of the attribute is true, meaning that expressions, **${...}**, are evaluated as dictated by the JSP specification. If the attribute is set to false, then expressions are not evaluated but rather treated as static text.

Following directive sets an expression not to be evaluated:

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

The isScriptingEnabled Attribute

The **isScriptingEnabled** attribute determines if the scripting elements are allowed for use.

The **default value (true)** enables scriptlets, expressions, and declarations. If the attribute's value is set to false, a translation-time error will be raised if the JSP uses any scriptlets, expressions (non-EL), or declarations.

The attribute’s value can be set to false if you want to restrict the usage of scriptlets, expressions (non-EL), or declarations:

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

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" />
```

JSP - 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 *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" />
```

Example

A good example of the **include** directive is including a common header and footer with multiple pages of content.

Let us define following three files **(a) header.jps**, **(b)footer.jsp**, and **(c)main.jsp** as follows:

Following is the content of **header.jsp**:

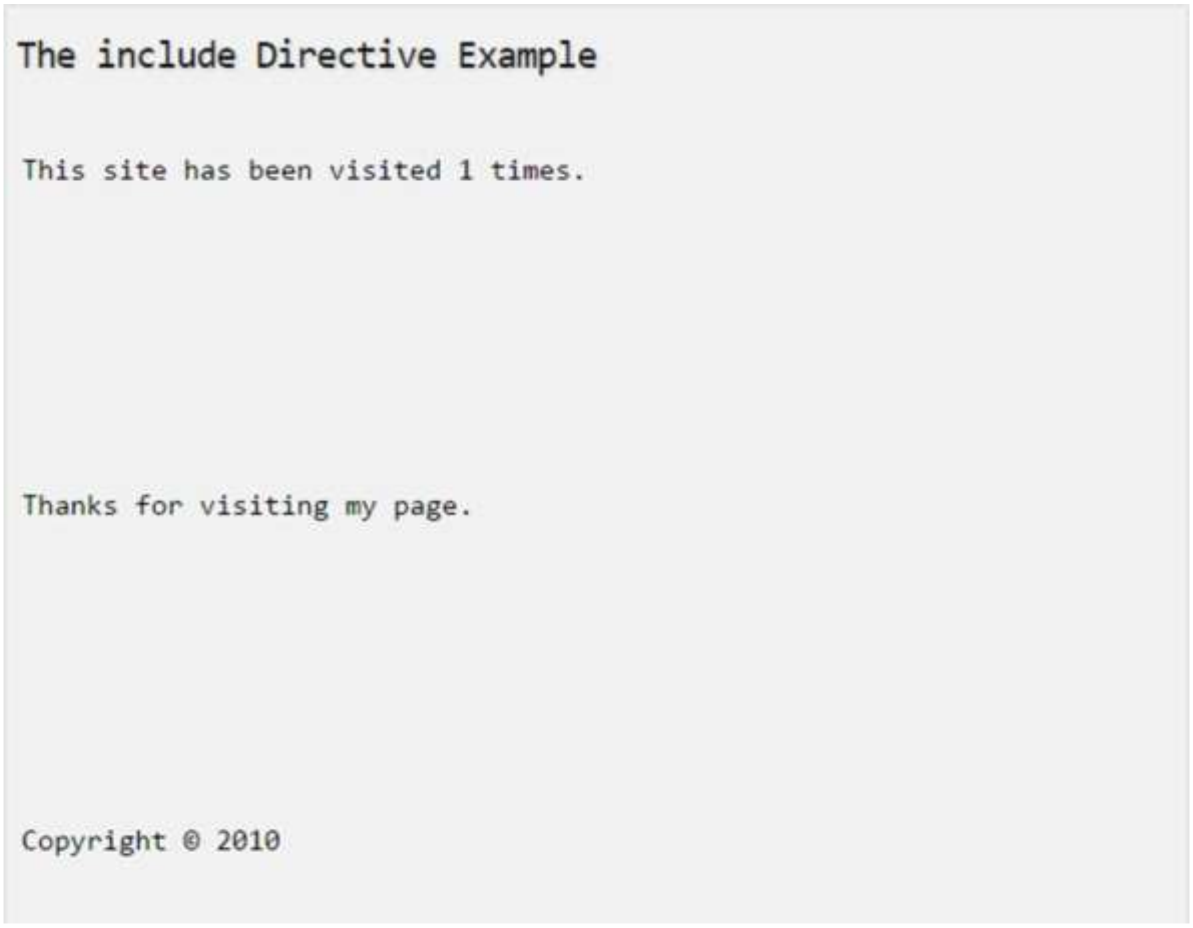
```
<%!
 int pageCount = 0;
 void addCount() {
   pageCount++;
 }
%>
<% addCount(); %>
<html>
<head>
<title>The include Directive Example</title>
</head>
<body>
<center>
<h2>The include Directive Example</h2>
<p>This site has been visited <%= pageCount %> times.</p>
</center>
<br/><br/>
```

Following is the content of **footer.jsp**:

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

```
<p>Thanks for visiting my page.</p>
</center>
<%@ include file="footer.jsp" %>
```

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 Directive Example

This site has been visited 1 times.

Thanks for visiting my page.

Copyright © 2010
```

Refresh **main.jsp** and you will find that the page hit counter keeps increasing.

You can design your webpages based on your creative instincts; it is recommended you keep the dynamic parts of your website in separate files and then include them in the main file. This makes it easy when you need to change a part of your webpage.

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" />
```

JSP - The taglib Directive

The JavaServer Pages API allows you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior.

The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides a means for identifying the custom tags in your JSP page.

The taglib directive follows the syntax given below:

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

Where the **uri** attribute value resolves to a location the container understands and the **prefix** attribute informs a container what bits of markup are custom actions.

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

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

When you use a custom tag, it is typically of the form **<prefix:tagname>**. The prefix is the same as the prefix you specify in the taglib directive, and the tagname is the name of a tag implemented in the tag library.

Example

For example, suppose the **custlib** tag library contains a tag called **hello**. If you wanted to use the hello tag with a prefix of **mytag**, your tag would be **<mytag:hello>** and it will be used in your JSP file as follows:

```
<%@ taglib uri="http://www.example.com/custlib" prefix="mytag" %>
<html>
```

```
<body>
<mytag:hello/>
</body>
</html>
```

We can call another piece of code using **<mytag:hello>**