**Introduction to Java Server Pages**

**(WEB DEVELOPMENT)**

**JSP** is a server side technology that does all the processing at server. It is used for creating dynamic web applications, using java as programming language.

Basically, any html file can be converted to JSP file by just changing the file extension from “.html” to “.jsp”, it would run just fine. What differentiates JSP from HTML is the ability to use java code inside HTML. In JSP, you can embed Java code in HTML using JSP tags for e.g. run the code below, every time you run this, it would display the current time. That is what makes this code dynamic.

Your First JSP

<HTML>

<HEAD>

<TITLE>MESSAGE</TITLE>

</HEAD>

<BODY>

<%out.print("Hello, Sample JSP code");%>

</BODY>

</HTML>

The **static content** can have text-based formats such as HTML,  XML etc and the **dynamic content** is generated by JSP tags using java code inside HTML.

## Servlet Vs JSP

Like JSP, Servlets are also used for generating dynamic webpages. Here is the comparison between them.

The major difference between them is that servlet adds HTML code inside java while JSP adds java code inside HTML. There are few other noticeable points that are as follows:  
**Servlets** –

1. Servlet is a Java program which supports HTML tags too.
2. Generally used for developing business layer(the complex computational code) of an enterprise application.
3. Servlets are created and maintained by Java developers.

**JSP** –

1. JSP program is a HTML code which supports java statements too. To be more precise, JSP embed java in html using JSP tags.
2. Used for developing  presentation layer of an enterprise application
3. Frequently used for designing websites and used by web developers.

## Advantages of JSP

1. JSP has all the advantages of servlet, like: Better performance than CGI Built in session features,  it also inherits the features of java technology like – multithreading, exception handling, Database connectivity, etc.
2. JSP Enables the separation of content generation from content presentation. Which makes it more flexible.
3. With the JSP, it is now easy for web designers to show case the information what is needed.
4. Web Application Programmers can concentrate on how to process/build the information.

## Architecture of a JSP Application

## Before we start developing web application, we should have a basic idea of architectures. Based on the location where request processing happens (Servlet OR JSP (java server pages)) there are two architectures for JSP. They are – Model1 Architecture & Model2 Architecture.

1. **Model1 Architecture**: In this Model, JSP plays a key role and it is responsible for of processing the request made by client. Client (Web browser) makes a request, JSP then creates a bean object which then fulfils the request and pass the response to JSP. JSP then sends the response back to client. Unlike Model2 architecture, in this Model most of the processing is done by JSP itself.

**JSP**

B

R

O

W

S

E

R

REQUEST

RESPONSE

DB

1. **Model2 Architecture (MVC)**: In this Model, Servlet plays a major role and it is responsible for processing the client’s (web browser) request. Presentation part (GUI part) will be handled by JSP and it is done with the help of bean as shown in image below. The servlet acts as controller and in charge of request processing. It creates the bean objects if required by the JSP page and calls the respective JSP page. The JSP handles the presentation part by using the bean object. In this Model, JSP doesn’t do any processing, Servlet creates the bean Object and calls the JSP program as per the request made by client.

CONTROLLER

**DB**

**JSP**

**SERVLET**

**B**

**R**

**O**

**W**

**S**

**E**

**R**

REQUEST

VIEW

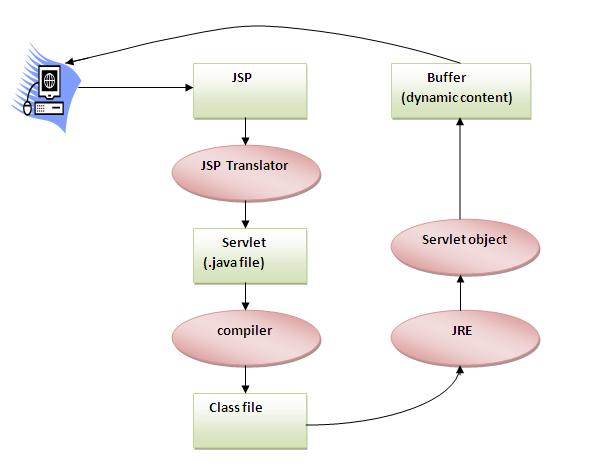
RESPONSE

# Java Server Pages (JSP) Life Cycle

When client makes a request to Server, it first goes to container. Then container checks whether the servlet class is older than jsp page(  To ensure that the JSP file got modified). If this is the case then container does the translation again (converts JSP to Servlet) otherwise it skips the translation phase (i.e. if JSP webpage is not modified then it doesn’t do the translation to improve the performance as this phase takes time and to repeat this step every time is not time feasible)

**The steps in the life cycle of jsp page are:**

* Translation of JSP Page
* Compilation of JSP Page
* Classloading (class file is loaded by the classloader)
* Instantiation (Object of the Generated Servlet is created).
* Initialization ( jspInit() method is invoked by the container).
* Reqeust processing ( \_jspService() method is invoked by the container).
* Destroy ( jspDestroy() method is invoked by the container).



As depicted in the above diagram, JSP page is translated into servlet by the help of JSP translator. The JSP translator is a part of webserver that is responsible to translate the JSP page into servlet. After that Servlet page is compiled by the compiler and gets converted into the class file. Moreover, all the processes that happens in servlet is performed on JSP later like initialization, committing response to the browser and destroy.

# JSP Declaration tag

Declaration tag is a block of java code for declaring class wide variables, methods and classes. Whatever placed inside these tags gets initialized during JSP initialization phase and has class scope. JSP container keeps this code outside of the service method (\_jspService()) to make them class level variables and methods.

As we know that variables can be initialized using [scriptlet](https://beginnersbook.com/2013/05/jsp-tutorial-scriptlets/) too but those declaration being placed inside \_jspService() method which doesn’t make them class wide declarations. On the other side, **declaration tag** can be used for defining class level variables, methods and classes.

**Syntax of declaration tag:**

<%!  Declaration %>

In this example we have declared two variables inside declaration tag and displayed them on client using **expression tag**.

<html>

<head>

<title>Declaration tag Example1</title>

</head>

<body>

<%! String name="Chaitanya"; %>

<%! int age=27; %>

<%= "Name is: "+ name %><br>

<%= "AGE: "+ age %>

</body>

</html>

**Example 2: Methods declaration**

In this example we have declared a method **sum** using **JSP declaration tag**.

<html>

<head>

<title>Methods Declaration</title>

</head>

<body>

<%!

int sum(int num1, int num2, int num3){

return num1+num2+num3;

}

%>

<%= "Result is: " + sum(10,40,50) %>

</body>

</html>

# JSP Expression Tag

Expression tag evaluates the expression placed in it, converts the result into String and send the result back to the client through **response object**. Basically it writes the result to the client (browser).

**Syntax of expression tag in JSP:**

<%= expression %>

## JSP expression tag Examples

**Example 1: Expression of values**

Here we are simply passing the expression of values inside expression tag.

<html>

<head>

<title>JSP expression tag example1</title>

</head>

<body>

<%= 2+4\*5 %>

</body>

</html>

**Example 2: Expression of variables**

In this example we have initialized few variables and passed the expression of variables in the expression tag for result evaluation.

<html>

<head>

<title>JSP expression tag example2</title>

</head>

<body>

<%

int a=10;

int b=20;

int c=30;

%>

<%= a+b+c %>

</body>

</html>

**Example 3: String and implicit object output**

In this example we are setting up an attribute using **application implicit object** and then displaying that attribute and a simple string on another JSP page using expression tag.

index.jsp

<html>

<head>

<title> JSP expression tag example3 </title>

</head>

<body>

<% application.setAttribute("MyName", "Chaitanya"); %>

<a href="display.jsp">Click here for display</a>

</body>

</html>

display.jsp

<html>

<head>

<title>Display Page</title>

</head>

<body>

<%="This is a String" %><br>

<%= application.getAttribute("MyName") %>

</body>

</html>

# JSP Scriptlets

**Scriptlets are nothing but java code** enclosed within **<% and %> tags**. JSP container moves the statements enclosed in it to **\_jspService()** method while generating servlet from JSP. The reason of copying this code to service method is: For each client’s request the \_jspService() method gets invoked, hence the code inside it executes for every request made by client.

**Syntax of Scriptlet:**

[code language=”java”]<% Executable java code %>[/code]

### JSP to Servlet transition for Scriptlet –

JSP doesn’t get executed directly, it first gets converted into a Servlet and then Servlet execution happens as normal. Also, I explained in first para that while translation from JSP to servlet, the java code is copied from scriptlet to \_jspService() method. Lets see how that happens.

**Sample JSP code:**

[code language=”html”]  
<H3> Sample JSP </H3>  
<% myMethod();%>  
[/code]

**Note:** Semicolon at the end of scriptlet.

**Corresponding translated Servlet code for above JSP code:**

[code language=”java”]  
public void \_jspService(HttpServletRequest request,  
HttpServletResponse response)  
throws ServletException, IOException {  
response.setContentType("text/html");  
HttpSession session = request.getSession();  
JspWriter out = response.getWriter();  
out.println("<H2>Sample JSP</H2>");  
myMethod();  
}[/code]

### **An example to learn about Scripting elements:**

[code language=”java”]  
<%– A jsp example to learn the JSP scripting elements–%>  
<%String string1 ="JSP scriptlet";%>  
<%!String string2 ="";%>  
<html>  
<head>  
<title> JSP page: Welcome </title>  
</head>  
<body>  
<h1>  
<%–This is an Expression statement–%>  
Welcome to <%=string1%>  
</h1>

<%–sciptlet example–%>  
<%if(localstring.equals("JSP scriptlet")){%>  
Hi  
<%}

else {%>  
hello  
<%} %>

<%–same thing can be done in this way also–%>  
<%if(localstring.equals("JSP scriptlet"))  
out.println("Hi"+string2);  
else  
out.println("hello");  
%>  
</body>  
</html>[/code]

**An example to show use of if -else using scriptlet –**  
Suppose there is a variable num and you want to display “hi” on your webpage if it is greater than 5 otherwise you wanna display a message. Consider the below code for this scenario –  
**If you wanna write a code in java for above situation then it would look like this –**

[code language=”java”]  
if (num > 5)  
{  
out.println("hi");  
}  
else  
{  
out.println("num value should not be less than 6");  
}[/code]

**To write the similar code in JSP we need to use JSP scriptlets – Code would be like this –**

[code language=”html”]  
<! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML4.0 translation //EN">  
<HTML>  
<HEAD>  
<TITLE> MY JSP PAGE </TITLE>  
</HEAD>  
<BODY>  
<% if (num > 5) { %>  
<H3> hi </H3>  
<%} else {%>  
<h3> num value should not be less than 6 </h3>  
<% } %>  
</BODY>  
</HTML>  
[/code]

**Important Point to remember:**Since the code inside it is a java code it **must end with a semicolon(;).**Now notice all the statements – you may find that  few scriptlet where we give semicolon in java, needs it here too and ends with a semicolon.

# JSP Directives – Page, Include and TagLib

Directives control the processing of an entire JSP page. It gives directions to the server regarding processing of a page.

**Syntax of Directives:**

<%@ directive name [attribute name=“value” attribute name=“value” ........]%>

**There are three types of Directives in JSP:**  
1) Page Directive  
2) Include Directive  
3) TagLib Directive

## 1) Page Directive

There are several attributes, which are used along with Page Directives and these are –

1. import
2. session
3. isErrorPage
4. errorPage
5. ContentType
6. isThreadSafe
7. extends
8. info
9. language
10. autoflush
11. buffer

### 1. import:

This attribute is used to import packages. While doing coding you may need to include more than one packages, In such scenarios this page directive’s attribute is very useful as it allows you to mention more than one packages at the same place separated by commas (,). Alternatively you can have multiple instances of page element each one with different package.

**Syntax of import attribute –**

<%@page import="value"%>

Here value is package name.

**Example of import-**The following is an example of how to import more than one package using import attribute of page directive.

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

<%@page import="java.lang.\*%>

<%--Comment: OR Below Statement: Both are Same--%>

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

### 2. session:

Generally while building a user interactive JSP application, we make sure to give access to the user to get hold of his/her personal data till the session is active. Consider an example of logging in into your bank account, we can access all of your data till we signout (or session expires). In order to maintain session for a page the session attribute should be true.

This attribute is to handle HTTP sessions for JSP pages. It can have two values: true or false. Default value for session attribute is true, which means if you do not mention this attribute, server may assume that HTTP session is required for this page.

**Default value for this attribute:**true

**Syntax of session attribute:**

<%@ page session="value"%>

here value is **either true OR false**

**Examples of session:**

<%@ page session="true"%>

The above code would allow a page to have session implicit objects.

<%@ page session="false"%>

If this code is specified in a JSP page, it means session objects will not be available for that page. Hence session cannot be maintained for that page.

### 3. isErrorPage:

This attribute is used to specify whether the current JSP page can be used as an error page for another JSP page. If value of isErrorPage is true it means that the page can be used for exception handling for another page. Generally these pages has error/warning messages OR exception handling codes and being called by another JSP page when there is an exception occurred there.

There is another use of isErrorPage attribute – The **exception implicit object** can only be available to those pages which has isErrorPage set to true. If the value is false, the page cannot use exception implicit object.

**Default value:**false

**Syntax of isErrorPage attribute:**

<%@ page isErrorPage="value"%>

Here value is either true OR false.

**Example of isErrorPage:**

<%@ page isErrorPage="true"%>

This makes a JSP page, a exception handling page.

### 4. errorPage:

When isErrorPage attribute is true for a particular page then it means that the page can be called by another page in case of an exception.  errorPage attribute is used to specify the URL of a JSP page which has isErrorPage attrbute set to true. It  handles the un-handled exceptions in the page.

**Syntax of errorPage attribute:**

<%@ page errorPage="value"%>

Here value is a JSP page name which has exception handling code (and isErrorPage set to true).

**Example of errorPage:**

<%@ page errorPage="ExceptionHandling.jsp"%>

This means if any exception occurs on the JSP page where this code has been placed, the ExceptionHandling.jsp (this page should have isErrorPage true) page needs to be called.

### 5. contentType:

This attribute is used to set the content type of a JSP page.

**Default value**: text/html

**Syntax of contentType attribute:**

<%@ page contentType="value"%>

here value of content type can be anything such as: text/html, text/xml etc.

**Example of contentType:**

Below code can be used for text/html pages.

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

for text/xml based pages:

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

### 6. isThreadSafe:

Lets understand this with an example. Suppose you have created a JSP page and mentioned isThreadSafe as true, it means that the JSP page supports multithreading (more than one thread can execute the JSP page simultaneously). On the other hand if it is set to false then JSP engine won’t allow multithreading which means only single thread will execute the page code.

**Default value for isThreadSafe attribute:** true.

**Syntax of isThreadSafe attribute:**

<%@ page isThreadSafe="value"%>

here value can be true OR false.

**Example of isThreadSafe:**

<%@ page isThreadSafe="false"%>

Only one thread will be responsible for JSP page execution.

### 7. buffer:

This attribute is used to specify the buffer size. If you specify this to none during coding then the output would directly written to Response object by JSPWriter. And, if you specify a buffer size then the output first written to buffer then it will be available for response object.

**Syntax of buffer attribute:**

<%@ page buffer="value"%>

value is **size in kb** or **none**.

**Example of buffer:**

No buffer for this page:

<%@ page buffer="none"%>

5 kb buffer size for the page, which has below code:

<%@ page buffer="5kb"%>

### 8. extends:

Like java, here also this attribute is used to extend (inherit) the class.

**Syntax of extends attribute:**

<%@ page extends="value"%>

Value is package\_name.class\_name.

**Example of extends:**

The below code will inherit the SampleClass from package: mypackage

<%@ page extends="mypackage.SampleClass"%>

### 9. info:

It provides a description to a JSP page. The string specified in info will return when we will call  getServletInfo() method.

**Syntax of info:**

<%@ page info="value"%>

here value is Message or Description

**Example of info attribute:**

<%@ page info="This code is given by Chaitanya Singh"%>

### 10. language:

It specifies the scripting language( underlying language) being used in the page.

**Syntax of language:**

<%@ page language="value"%>

value is scripting language here.

**Example of language attribute:**

<%@ page language="java"%>

### 11. autoFlush:

If it is true it means the buffer should be flushed whenever it is full. false will throw an exception when buffer overflows.

**Default value**: True

**Syntax of autoFlush:**

<%@ page autoFlush="value"%>

value can be true or false.

**Example of autoFlush attribute:**

Buffer will be flushed out when it is full –

<%@ page autoFlush="true"%>

It will throw an exception when buffer is full due to overflow condition

<%@ page autoFlush="true"%>

### 12. isScriptingEnabled:

It has been dropped and not in use.

### 13. isELIgnored:

This attribute specify whether expressions will be evaluated or not.

**Default value**: true

**Syntax of isELIgnored:**

<%@ page isELIgnored="value"%>

value can be true or false.

**Example of isELIgnored attribute:**

Any expression present inside JSP page will not be evaluated –

<%@ page isELIgnored="false"%>

Expression will be evaluated (true is a default value so no need to specify)-

<%@ page isELIgnored="true"%>

## 2) Include Directive

Include directive is used to copy the content of one JSP page to another. It’s like including the code of one file into another.

**Syntax of Include Directive:**

<%@include file ="value"%>

here value is the JSP file name which needs to be included. If the file is in the same directory then just specify the file name otherwise complete URL(or path) needs to be mentioned in the value field.

**Note: It can be used anywhere in the page.**

**Example:**

<%@include file="myJSP.jsp"%>

You can use the above code in your JSP page to copy the content of myJSP.jsp file. However in this case both the JSP files must be in the same directory. If the myJSP.jsp is in the different directory then instead of just file name you would need to specify the complete path in above code.

## 3) Taglib Directive

This directive basically allows user to use Custom tags in JSP. Taglib directive helps you to declare custom tags in JSP page.

**Syntax of Taglib Directive:**

<%@taglib uri ="taglibURI" prefix="tag prefix"%>

Where URI is uniform resource locator, which is used to identify the location of custom tag and tag prefix is a string which can identify the custom tag in the location identified by uri.

**Example of Targlib:**

<%@ taglib uri="http://www.sample.com/mycustomlib" prefix="demotag" %>

<html>

<body>

<demotag:welcome/>

</body>

</html>

As you can see that uri is having the location of custom tag library and prefix is identifying the prefix of custom tag.

**Note**: In above example – <demotag: welcome> has a prefix demotag.

# Include Directive in JSP

**Include directive** is used for merging external files to the current JSP page during translation phase (The phase where JSP gets converted into the equivalent Servlet).

**Why we need to use the include directive? Can’t we simply add the file’s content in the current JSP instead of using the directive?**

We can copy the content of external file and paste it in the main JSP, however it would **not be a good practice**. Let’s understand this with the help of an example – I have 100 external files and 1 main JSP file. If I just copy the content of all files in the main JSP then I have to edit it whenever there is a change in any of the external file, instead we can include all files using directive and edit the particular file whenever needed.

Also, by using include directive you can enhance the**code re-usability** – Suppose there is a certain code or data which needs to be there in all the JSP pages of your application then you can simply have that code/data in one file and include the file in all the JSP pages.

The above two reasons can be considered as advantages of using include directive.

### **Syntax:**

This is the syntax of **include directive in JSP**.

<%@ include file="URL of the file" %>

We must need to specify relative URL –

If file is in the same folder where the current JSP page resides then we can just mention the file name else the relative path to the file needs to be specified.

### Include Directive Example

index.jsp

<html>

<head>

<title>Main JSP Page</title>

</head>

<body>

<%@ include file="file1.jsp" %>

Main JSP Page: Content between two include directives.

<%@ include file="file2.jsp" %>

</body>

</html>

file1.jsp

<p align="center">

This is my File1.jsp and I will include it in index.jsp using include directive

</p>

file2.jsp

<p align="center">

This is File2.jsp

</p>

# JSP include directive with parameters example

We discussed **JSP include action with parameters**. Here we will see how to pass parameters when using **JSP include directive**.

## Example

In this example we are passing three string parameters to the included JSP page.

index.jsp

<%@ include file="file1.jsp" %>

<html>

<head>

<title>Passing Parameters to Include directive</title>

</head>

<body>

<%!

String country="India";

String state="UP";

String city="Agra";

%>

<%

session.setAttribute("co", country);

session.setAttribute("st", state);

session.setAttribute("ci", city);

%>

</body>

</html>

Above, I have used **declaration tag** for initializing strings and **scriptlet** for setting up them in **session object**. As the use of sciptlet is disregarded a long back, alternatively you can use **<c:set> JSTL tag** for doing the same – The code would be like this –

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

<c:set var="co" value="India" scope="session"/>

<c:set var="st" value="UP" scope="session"/>

<c:set var="ci" value="Agra" scope="session"/>

<%@ include file="file1.jsp" %>

file1.jsp

<%=session.getAttribute("co") %>

<%=session.getAttribute("st") %>

<%=session.getAttribute("ci") %>

In the above example I have passed the parameters using session implicit object, however you can also pass them using request, page and application implicit objects.

# Exception handling in JSP

Before going through **exception handling in JSP**, let’s understand what is exception and how it is different from errors.

**Exception:**These are nothing but the abnormal conditions which interrupts the normal flow of execution. Mostly they occur because of the wrong data entered by user. It is must to handle exceptions in order to give meaningful message to the user so that user would be able to understand the issue and take appropriate action.

**Error:** It can be a issue with the code or a system related issue. We should not handle errors as they are meant to be fixed.

**Methods of handling exceptions:**

We can handle exceptions using the below two methods.

* Exception handling using exception **implicit object**
* Exception handling using try catch blocks within scriptlets

### Exception handling using exception implicit object

We have specified the exception handling page using **errorPage** attribute of **Page directive**. If any exception occurs in the main JSP page the control will be transferred to the page mentioned in errorPage attribute.

The handler page should have **isErrorPage set to true** in order to use **exception implicit object**. That’s the reason we have set the isErrorPage true for errorpage.jsp.

index.jsp

<%@ page errorPage="errorpage.jsp" %>

<html>

<head>

<title>JSP exception handling example</title>

</head>

<body>

<%

//Declared and initialized two integers

int num1 = 122;

int num2 = 0;

//It should throw Arithmetic Exception

int div = num1/num2;

%>

</body>

</html>

errorpage.jsp

<%@ page isErrorPage="true" %>

<html>

<head>

<title>Display the Exception Message here</title>

</head>

<body>

<h2>errorpage.jsp</h2>

<i>An exception has occurred in the index.jsp Page.

Please fix the errors. Below is the error message:</i>

<b><%= exception %></b>

</body>

</html>

### Exception handling using try catch blocks within scriptlets

We have handled the exception using try catch blocks in the below example. Since try catch blocks are java code so it must be placed inside sciptlet. In the below example I have declared an array of length 5 and tried to access the 7th element which doesn’t exist. It caused Array Index out of bounds exception.

error.jsp

<html>

<head>

<title>Exception handling using try catch blocks</title>

</head>

<body>

<%

try{

//I have defined an array of length 5

int arr[]={1,2,3,4,5};

//I'm assinging 7th element to int num

//which doesn't exist

int num=arr[6];

out.println("7th element of arr"+num);

}

catch (Exception exp){

out.println("There is something wrong: " + exp);

}

%>

</body>

</html>

# JSP Actions – Java Server Pages

## Directives vs Actions

1. Directives are used during translation phase while actions are used during request processing phase.
2. Unlike Directives Actions are re-evaluated each time the page is accessed.

The following are the action elements used in JSP:

## 1. <jsp:include> Action

Like **include page directive**this action is also used to insert a JSP file in another file.

**<jsp:include> vs include directive :**

It has the same difference which I mentioned at the beginning of the article (directive vs action). In <jsp:include> the file is being included during request processing while in case of include directive it has been included at translation phase.

**Syntax of <jsp:include> :**

<jsp:include page="page URL"  flush="Boolean Value" />

Here page URL is: the location of the page needs to be included & flush value can be either true or false (Boolean value).

**Example:**

<html>

<head>

<title>Demo of JSP include Action Tag</title>

</head>

<body>

<h3>JSP page: Demo Include</h3>

<jsp:include page="sample.jsp" flush="false" />

</body>

</html>

**page**: Page value is sample.jsp which means this is the page needs to be included in the current file. Just the file name mentioned which shows that the sample.jsp is in the same directory.

**flush**: Its value is false, which means resource buffer has not been flushed out before including to the current page.

## 2. <jsp:forward> Action

<jsp:forward> is used for redirecting the request. When this action is encountered on a JSP page the control gets transferred to the page mentioned in this action.

**Syntax of <jsp:forward> :**

<jsp:forward page="URL of the another static, JSP OR Servlet page" />

**Example:**  
**first.jsp**

<html>

<head>

<title>Demo of JSP Forward Action Tag</title>

</head>

<body>

<h3>JSP page: Demo forward</h3>

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

</body>

</html>

Now when JSP engine would execute first.jsp (the above code) then after action tag, the request would be transferred to another JSP page (second.jsp).

**Note**: first.jsp and second.jsp should be in the same directory otherwise you have to specify the complete path of second.jsp.

## 3. <jsp:param> Action

This action is useful for passing the parameters to Other JSP action tags such as JSP include & JSP forward tag. This way new JSP pages can have access to those parameters using request object itself.

**Syntax of <jsp:param>:**

<jsp: param name="param\_name\_here" value="value\_of\_parameter\_here" />

Now considers the same above example –

**first.jsp**

<html>

<head>

<title>Demo of JSP Param Action Tag</title>

</head>

<body>

<h3>JSP page: Demo Param along with forward</h3>

<jsp:forward page="second.jsp">

<jsp:param name ="date" value="20-05-2012" />

<jsp:param name ="time" value="10:15AM" />

<jsp:param name ="data" value="ABC" />

</jsp:forward>

</body>

</html>

In the above example first.jsp is passing three parameters (data, time & data) to second.jsp and second.jsp can access these parameters using the below code –

Date:<%= request.getParameter("date") %>

Time:<%= request.getParameter("time") %>

My Data:<%= request.getParameter("data") %>

## 4. <jsp:useBean> Action

This action is useful when you want to use Beans in a JSP page, through this tag you can easily invoke a bean.

**Syntax of <jsp:useBean>:**

<jsp: useBean id="unique\_name\_to\_identify\_bean"  class="package\_name.class\_name" />

**Example of <jsp:useBean>, <jsp:setProperty> & <jsp:getProperty>:**

Once Bean class is instantiated using above statement, you have to use **jsp:setProperty** and **jsp:getProperty** actions to use the bean’s parameters. we will see both **setProperty** and **getProperty** after this action tag.

**StudentBeanTest.jsp**

<html>

<head>

<title>JSP Page to show use of useBean action</title>

</head>

<body>

<h1>Demo: Action</h1>

<jsp:useBean id="student" class="javabeansample.StuBean"/>

<jsp:setProperty name="student" property="\*"/>

<h1>

name:<jsp:getProperty name="student" property="name"/><br>

empno:<jsp:getProperty name="student" property="rollno"/><br>

</h1>

</body>

</html>

**StudentBean.java**

package javabeansample;

public class StuBean {

public StuBean() {

}

private String name;

private int rollno;

public void setName(String name)

{

this.name=name;

}

public String getName()

{

return name;

}

public void setRollno(int rollno)

{

this.rollno=rollno;

}

public int getRollno()

{

return rollno;

}

}

## 5. <jsp:setProperty> Action

This action tag is used to set the property of a Bean, while using this action tag, you may need to specify the Bean’s unique name (it is nothing but the id value of useBean action tag).

**syntax of <jsp:setProperty>**

<jsp: useBean id="unique\_name\_to\_identify\_bean"  class="package\_name.class\_name" />

....

....

<jsp:setProperty name="unique\_name\_to\_identify\_bean" property="property\_name" />

OR

<jsp: useBean id="unique\_name\_to\_identify\_bean"  class="package\_name.class\_name">

....

....

<jsp:setProperty name="unique\_name\_to\_identify\_bean" property="property\_name" />

</jsp:useBean>

In property\_name, you can also use ‘\*’, which means any request parameter which matches to the Bean’s property will be passed to the corresponding setter method.

## 6. <jsp:getProperty> Action

It is used to retrieve or fetch the value of Bean’s property.

**syntax of <jsp:getProperty>**

<jsp: useBean id="unique\_name\_to\_identify\_bean"  class="package\_name.class\_name" />

....

....

<jsp:getProperty name="unique\_name\_to\_identify\_bean" property="property\_name" />

OR

<jsp: useBean id="unique\_name\_to\_identify\_bean"  class="package\_name.class\_name">

....

....

<jsp:getProperty name="unique\_name\_to\_identify\_bean" property="property\_name" />

</jsp:useBean>

## Other Action Tags

The below action tags are not frequently used so I haven’t covered them in detail.

### 7. <jsp:plugin> Action

This tag is used when there is a need of a plugin to run a Bean class or an Applet.

[http://docs.oracle.com/javase/7/docs/technotes/guides/plugin/developer\_guide/jsp.html](https://docs.oracle.com/javase/7/docs/technotes/guides/plugin/developer_guide/jsp.html)

### 8. <jsp:body> Action

### 9. <jsp:element> Action

### 10. <jsp:text> Action

### 11. <jsp:attribute> Action

# JSP include action tag

Include action tag is used for including another resource to the current JSP page. The included resource can be a static page in HTML, JSP page or Servlet. We can also pass parameters and their values to the resource which we are including. Below I have shared two examples of <jsp:include>, one which includes a page without passing any parameters and in second example we are passing few parameters to the page which is being included.

**Syntax:**

1) Include along with parameters.

<jsp:include page="Relative\_URL\_Of\_Page">

<jsp:param ... />

<jsp:param ... />

<jsp:param ... />

...

<jsp:param ... />

</jsp:include>

2) Include of another resource without sharing parameters.

<jsp:include page="Relative\_URL\_of\_Page" />

Relative\_URL\_of\_Page would be the page name if the page resides in the same directory where the current JSP resides.

### Example 1: <jsp:include> without parameters

In this example we will use <jsp:include> action tag without parameters. As a result the page will be included in the current JSP page as it is:

index.jsp

<html>

<head>

<title>JSP Include example</title>

</head>

<body>

<b>index.jsp Page</b><br>

<jsp:include page="Page2.jsp" />

</body>

</html>

Page2.jsp

<b>Page2.jsp</b><br>

<i> This is the content of Page2.jsp page</i>

Content of Page2.jsp has been appended in the index.jsp.

### Example 2: Use of <jsp:include> along with <jsp:param>

index.jsp

I’m using **<jsp:include> action** here along with <jsp:param> for passing parameters to the page which we are going to include.

<html>

<head>

<title>JSP Include example with parameters</title>

</head>

<body>

<h2>This is index.jsp Page</h2>

<jsp:include page="display.jsp">

<jsp:param name="userid" value="Chaitanya" />

<jsp:param name="password" value="Chaitanya" />

<jsp:param name="name" value="Chaitanya Pratap Singh" />

<jsp:param name="age" value="27" />

</jsp:include>

</body>

</html>

display.jsp

<html>

<head>

<title>Display Page</title>

</head>

<body>

<h2>Hello this is a display.jsp Page</h2>

UserID: <%=request.getParameter("userid") %><br>

Password is: <%=request.getParameter("password") %><br>

User Name: <%=request.getParameter("name") %><br>

Age: <%=request.getParameter("age") %>

</body>

</html>

**Output:**

As you can see that the content of display.jsp has been included in index.jsp. Also, the parameters we have passed are displaying correctly in the included page.

# JSP forward action tag

JSP forward action tag is used for forwarding a request to the another resource (It can be a JSP, static page such as html or Servlet). Request can be forwarded with or without parameter. In this tutorial we will see examples of  <jsp:forward> action tag.

**Syntax:**

1) Forwarding along with parameters.

<jsp:forward page="display.jsp">

<jsp:param ... />

<jsp:param ... />

<jsp:param ... />

...

<jsp:param ... />

</jsp:forward>

2) Forwarding without parameters.

<jsp:forward page="Relative\_URL\_of\_Page" />

Relative\_URL\_of\_Page: If page is in the same directory where the main page resides then use page name itself as I did in the below examples.

### JSP Forward Example 1 – without passing parameters

In this example we are having two JSP pages – index.jsp and display.jsp. We have used <jsp:forward> action tag in index.jsp for forwarding the request to display.jsp. Here we are not passing any parameters while using the action tag. In the next example we will pass the parameters as well to another resource.

index.jsp

<html>

<head>

<title>JSP forward action tag example</title>

</head>

<body>

<p align="center">My main JSP page</p>

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

</body>

</html>

display.jsp

<html>

<head>

<title>Display Page</title>

</head>

<body>

Hello this is a display.jsp Page

</body>

</html>

**Output:**

Below is the output of above cpde. It is basically the content of display.jsp, which clearly shows that index.jsp didn’t display as it forwarded the request to the **display.jsp** page.

### JSP Forward Example 2 – with parameters

Here we are passing the parameters along with forward request. For passing parameters we are using <jsp:param> action tag. In this example we are passing 4 parameters along with forward and later we are displaying them on the forwarded page. In order to fetch the parameters on display.jsp page we are using getParameter method of [request implicit object](https://beginnersbook.com/2013/11/jsp-implicit-object-request-with-examples/).

index.jsp

<html>

<head>

<title>JSP forward example with parameters</title>

</head>

<body>

<jsp:forward page="display.jsp">

<jsp:param name="name" value="Chaitanya" />

<jsp:param name="site" value="Facebook.com" />

<jsp:param name="tutorialname" value="jsp forward action" />

<jsp:param name="reqcamefrom" value="index.jsp" />

</jsp:forward>

</body>

</html>

display.jsp

<html>

<head>

<title>Display Page</title>

</head>

<body>

<h2>Hello this is a display.jsp Page</h2>

My name is: <%=request.getParameter("name") %><br>

Website: <%=request.getParameter("site") %><br>

Topic: <%=request.getParameter("tutorialname") %><br>

Forward Request came from the page: <%=request.getParameter("reqcamefrom") %>

</body>

</html>

**Output:**

Above code directly displayed display.jsp page, which is displaying the parameters passed from index.jsp page.

# jsp:useBean, jsp:setProperty and jsp:getProperty Action Tags

we will see how to use a bean class in JSP with the help of **jsp:useBean, jsp:setProperty and jsp:getProperty**action tags.

**Syntax of jsp:useBean:**

<jsp: useBean id="unique\_name\_to\_identify\_bean"

class="package\_name.class\_name" />

**Syntax of jsp:setProperty:**

<jsp:setProperty name="unique\_name\_to\_identify\_bean"

property="property\_name" />

**Syntax of jsp:getProperty:**

<jsp:getProperty name="unique\_name\_to\_identify\_bean"

property="property\_name" />

### A complete example of useBean, setProperty and getProperty

1) We have a bean class Details where we are having three variables username, age and password. In order to use the bean class and it’s properties in JSP we have initialized the class like this in the userdetails.jsp page –

<jsp:useBean id="userinfo" class="abc.com.Details"></jsp:useBean>

We have used useBean action to initialize the class. Our class is in abc.com package so we have given a fully qualified name .

2) We have mapped the properties of bean class and JSP using setProperty action tag. We have given ‘\*’ in the property field to map the values based on their names because we have used the same property name in bean class and index.jsp JSP page. In the name field we have given the unique identifier which we have defined in useBean tag.

<jsp:setProperty property="\*" name="userinfo"/>

3) To get the property values we have used getProperty action tag.

<jsp:getProperty property="*propertyname*" name="userinfo"/>

Details.java

public class Details {

public Details() {

}

private String username;

private int age;

private String password;

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

}

index.jsp

<html>

<head><title>

useBean, getProperty and setProperty example

</title></head><body>

<form action="userdetails.jsp" method="post">

User Name: <input type="text" name="username"><br>

User Password: <input type="password" name="password"><br>

User Age: <input type="text" name="age"><br>

<input type="submit" value="register">

</form>

</body>

</html>

userdetails.jsp

<jsp:useBean id="userinfo" class="abc.com.Details"></jsp:useBean>

<jsp:setProperty property="\*" name="userinfo"/>

You have enterted below details:<br>

<jsp:getProperty property="username" name="userinfo"/><br>

<jsp:getProperty property="password" name="userinfo"/><br>

<jsp:getProperty property="age" name="userinfo" /><br>

# JSP Implicit Objects

These objects are created by JSP Engine during translation phase (while translating JSP to Servlet). They are being created inside service method so we can directly use them within **Scriptlet**without initializing and declaring them. There are total 9 implicit objects available in JSP.

**Implicit Objects and their corresponding classes:**

|  |  |
| --- | --- |
| out | javax.servlet.jsp.JspWriter |
| request | javax.servlet.http.HttpServletRequest |
| response | javax.servlet.http.HttpServletResponse |
| session | javax.servlet.http.HttpSession |
| application | javax.servlet.ServletContext |
| exception | javax.servlet.jsp.JspException |
| page | java.lang.Object |
| pageContext | javax.servlet.jsp.PageContext |
| config | javax.servlet.ServletConfig |

1. **Out**: This is used for writing content to the client (browser). It has several methods which can be used for properly formatting output message to the browser and for dealing with the buffer.
2. **Request**: The main purpose of request implicit object is to get the data on a JSP page which has been entered by user on the previous JSP page. While dealing with login and signup forms in JSP we often prompts user to fill in those details, this object is then used to get those entered details on an another JSP page (action page) for validation and other purposes.
3. **Response**: It is basically used for modfying or delaing with the response which is being sent to the client (browser) after processing the request.
4. **Session:** It is most frequently used implicit object, which is used for storing the user’s data to make it available on other JSP pages till the user session is active.
5. **Application:** This is used for getting application-wide initialization parameters and to maintain useful data across whole JSP application.
6. **Exception:** Exception implicit object is used in exception handling for displaying the error messages. This object is only available to the JSP pages, which has isErrorPage set to true.
7. **Page:** Page implicit object is a reference to the current Servlet instance (Converted Servlet, generated during translation phase from a JSP page). We can simply use **this** in place of it. I’m not covering it in detail as it is rarely used and not a useful implicit object while building a JSP application.
8. **pageContext**: It is used for accessing page, request, application and session attributes.
9. **Config:** This is a Servlet configuration object and mainly used for accessing getting configuration information such as servlet context, servlet name, configuration parameters etc.

# Session Implicit Object in JSP with examples

Session is most frequently used implicit object in JSP. The main usage of it to gain access to all the user’s data till the user session is active.

### Methods of session Implicit Object

1. **setAttribute(String, object) –**This method is used to save an object in session by assigning a unique string to the object. Later, the object can be accessed from the session by using the same String till the session is active. setAttribute and getAttribute are the two most frequently used methods while dealing with session in JSP.
2. **getAttribute(String name) –**The object stored by setAttribute method is fetched from session using getAttribute method. For example if we need to access userid on every jsp page till the session is active then we should store the user-id in session using setAttribute method and can be accessed using getAttribute method whenever needed.
3. **removeAttribute(String name) –**The objects which are stored in session can be removed from session using this method. Pass the unique string identifier as removeAttribute’s method.
4. **getAttributeNames –**It returns all the objects stored in session. Basically, it results in an enumeration of objects.
5. **getCreationTime –**This method returns the session creation time, the time when session got initiated (became active).
6. **getId –**Servlet container assigns a unique string identifier to session while creation of it. **getId**method returns that unique string identifier.
7. **isNew() –**Used to check whether the session is new. It returns Boolean value (true or false). Mostly used to track whether the cookies are enabled on client side. If cookies are disabled the session.isNew() method would always return **true**.
8. **invalidate() –**It kills a session and breaks the association of session with all the stored objects.
9. **getMaxInactiveInterval –**Returns session’s maximum inactivate time interval in seconds.
10. **getLastAccessedTime –**Generally used to know the last accessed time of a session.

### Session Implicit Object Example

The below html page would display a text box along with a submit button. The submit action would transfer the control to session.jsp page.

index.html

<html>

<head>

<title>Welcome Page: Enter your name</title>

</head>

<body>

<form action="session.jsp">

<input type="text" name="inputname">

<input type="submit" value="click here!!"><br/>

</form>

</body>

</html>

The **session.jsp** page displays the name which user has entered in the index page and it stores the the same variable in the **session object** so that it can be fetched on any page until the session becomes inactive.

session.jsp

<html>

<head>

<title>Passing the input value to a session variable</title>

</head>

<body>

<%

String uname=request.getParameter("inputname");

out.print("Welcome "+ uname);

session.setAttribute("sessname",uname);

%>

<a href="output.jsp">Check Output Page Here </a>

</body>

</html>

In this page we are fetching the variable’s value from session object and displaying it.

output.jsp

<html>

<head>

<title>Output page: Fetching the value from session</title>

</head>

<body>

<%

String name=(String)session.getAttribute("sessname");

out.print("Hello User: You have entered the name: "+name);

%>

</body>

</html>

# How to validate and invalidate session in JSP

We have already seen invalidate() method in [session implicit object](https://beginnersbook.com/2013/11/jsp-implicit-object-session-with-examples/) article. In this post we are going to discuss it in detail. Here we will see how to validate/invalidate a session.

### Example

Lets understand this with the help of an example: In the below example we have three jsp pages.

* index.jsp: It is having four variables which are being stored in session object.
* display.jsp: It is fetching the attributes (variables) from session and displaying them.
* errorpage.jsp: It is first calling session.invalidate() in order to invalidate (make the session inactive) the session and then it has a logic to validate the session (checking whether the session is active or not).

index.jsp

<%

String firstname="Chaitanya";

String middlename="Pratap";

String lastname="Singh";

int age= 26;

session.setAttribute( "fname", firstname );

session.setAttribute( "mname", middlename );

session.setAttribute( "lname", lastname );

session.setAttribute( "UAge", age );

%>

<a href="display.jsp">See Details</a>

<a href="errorpage.jsp">Invalidate Session</a>

display.jsp

<%= session.getAttribute( "fname" ) %>

<%= session.getAttribute( "mname" ) %>

<%= session.getAttribute( "lname" ) %>

<%= session.getAttribute( "UAge" ) %>

errorpage.jsp

<%session.invalidate();%>

<% HttpSession nsession = request.getSession(false);

if(nsession!=null) {

String data=(String)session.getAttribute( "fname" );

out.println(data);

}

else

out.println("Session is not active");

%>

**Output:**  
while opening display page, all the attributes are getting displayed on client (browser).  
Since we have already called invalidate in the first line of errorpage.jsp, it is displaying the message “Session is not active” on the screen

**Points to Note:**  
1) This will deactivate the session

<%session.invalidate();%>

2) This logic will exceute the if body when the session is active else it would run the else part.

<% HttpSession nsession = request.getSession(false);

if(nsession!=null)

...

else

...

%>

# Request Implicit Object in JSP with examples

It is mainly used to get the data on a JSP page which has been entered by user on the previous JSP page.

### Methods of request Implicit Object

1. **getParameter(String name)**– This method is used to get the value of a request’s parameter. For example at login page user enters user-id and password and once the credentials are verified the login page gets redirected to user information page, then using request.getParameter we can get the value of user-id and password which user has input at the login page.
2. String Uid= request.getParameter("user-id");

String Pass= request.getParameter("password");

1. **getParameterNames() –** It returns enumeration of all the parameter names associated to the request.

Enumeration e= request.getParameterNames();

1. **getParameterValues(String name) –**It returns the array of parameter values.

String[] allpasswords = request.getParameterValues("password");

1. **getAttribute(String name) –**Used to get the attribute value.  request.getAttribute(“admin”) would give you the value of attribute admin.
2. **getAttributeNames() –**It is generally used to get the attribute names associated to the current session. It returns the enumeration of attribute names present in session.

Enumerator e = request.getAttributeNames();

1. **setAttribute(String,Object) –**It assigns an object’s value to the attribute. For example I have an attribute **password**and a String object str which has a value **“admin”**then calling request.setAttribute(“password”, str) would assign a value **admin** to the attribute **password**.
2. **removeAttribute(String) –**By using this method a attribute can be removed and cannot be used further. For e.g. If you have a statement **request.removeAttribute(“userid”)** on a JSP page then the userid attribute would be completely removed and request.getAttribute(“userid”) would return **NULL** if used after the removeAttribute method.
3. **getCookies() –**It returns an array of cookie objects received from the client. This method is mainly used when dealing with cookies in JSP.
4. **getHeader(String name) –**This method is used to get the header information of the request.
5. **getHeaderNames() –**Returns enumerator of all header names. Below code snippet would display all the header names associated with the request.
6. Enumeration e = request.getHeaderNames();
7. while (enumeration.hasMoreElements()) {
8. String str = (String)e.nextElement();
9. out.println(str);

}

1. **getRequestURI() –**This method (request.getRequestURI()) returns the URL of current JSP page.
2. **getMethod() –** It returns HTTP request method. request.getMethod(). For example it will return GET for a Get request and POST for a Post Request.
3. **getQueryString() –**Used for getting the query string associated to the JSP page URL. It is the string associted to the URL after question mark sign (?).

### Request Implicit Object Example

In the below example we are receiving the input from user in index.html page and displaying the same information in userinfo.jsp page using request implicit object.

index.html

<html>

<head>

<title>Enter UserName and Password</title>

</head>

<body>

<form action="userinfo.jsp">

Enter User Name: <input type="text" name="uname" /> <br><br>

Enter Password: <input type="text" name="pass" /> <br><br>

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

</form>

</body>

</html>

userinfo.jsp

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

<html>

<body>

<%

String username=request.getParameter("uname");

String password=request.getParameter("pass");

out.print("Name: "+username+" Password: "+password);

%>

</body>

</html>

# Response Implicit Object in JSP with examples

We are going to discuss about response implicit object in JSP. It is an instance of **javax.servlet.http.HttpServletRequest** and mainly used for modifying the response which is being sent to the browser after processing the client’s request.

### Methods of response Implicit Object

void setContentType(String type)  
void sendRedirect(String address)  
void addHeader(String name, String value)  
void setHeader(String name, String value)  
boolean containsHeader(String name)  
void addCookie(Cookie value)  
void sendError(int status\_code, String message)  
boolean isCommitted()  
void setStatus(int statuscode)

Let’s see each method in detail –

1. **void setContentType(String type) –**This method tells browser, the type of response data by setting up the MIME type and character encoding. The information sets by this method helps browser to interpret the response. Example –
2. response.setContentType("text/html");
3. response.setContentType("image/gif");
4. response.setContentType("image/png");

response.setContentType("application/pdf");

1. **void sendRedirect(String address) –**It redirects the control to a new JSP page. For e.g. When the browser would detect the below statement, it would be redirected to the book.com from the current JSP page.

response.sendRedirect("http://book.com");

1. **void addHeader(String name, String value) –**addHeader method adds a header to the response, basically it includes a header name and it’s value. For example – The below statement will include a header “Site” in the response with value “Book.com”.

response.addHeader("Site", "Book.com");

1. **void setHeader(String name, String value) –**It sets the header value. This method overrides the current value of header with the new value.  Let’s say I’m modifying the value of Header “**Site**“. The below statement would modify the current value Book.com to a new value BB.com

response.setHeader("Site", "BB.com");

1. **boolean containsHeader(String name) –**It returns a Boolean value true/false. It basically checks the whether the header is present in the response or not. For example – Above, in the addHeader method example we have added a **Site** Header in response so the below statement would return true.

response.containsHeader("Site");

1. **void addCookie(Cookie cookie) –**This method adds a cookie to the response. The below statements would add 2 Cookies **Author**and**Siteinfo**to the response.
2. response.addCookie(Cookie Author);

response.addCookie(Cookie Siteinfo);

1. **void sendError(int status\_code, String message) –**It is used to send error response with a code and an error message. For example –

response.sendError(404, "Page not found error");

1. **boolean isCommitted()** -It checks whether the Http Response has been sent to the client, if yes then it returns true else it gives false.
2. <% if(response.isCommited())
3. {
4. <%--do something --%>
5. }else
6. {
7. <%--do something else --%>

} %>

1. **void setStatus(int statuscode)** – This method is used to set the HTTP status to a given value. For e.g. the below statement would set HTTP response code to 404 (Page not found).

response.setStatus(404);

### Response Implicit Object Example

In the below example we are receiving id and password from login page and then we are matching them with hardcoded correct id/pass. If the credentials are correct the sign-in page redirects to success page else it redirects to sign-in fail JSP page.

index.html

<html>

<head>

<title>Login Page</title>

</head>

<body>

<form action="checkdetails.jsp">

UserId: <input type="text" name="id" /> <br><br>

Password: <input type="text" name="pass" /> <br><br>

<input type="submit" value="Sign In!!"/>

</form>

</body>

</html>

This JSP page verifies the input id/pass against hard-coded values.

checkdetails.jsp

<html>

<head><title>Check Credentials</title>

</head>

<body>

<%

String uid=request.getParameter("id");

String password=request.getParameter("pass");

session.setAttribute("session-uid", uid);

if(uid.equals("Chaitanya") && password.equals("Book"))

{

response.sendRedirect("success.jsp");

}

else

{

response.sendRedirect("failed.jsp");

}

%>

</body>

</html>

This JSP page would execute if id/pass are matched to the hardcoded userid/password.

success.jsp

<html>

<head><title>Success Page</title>

</head>

<body>

<%

String data=(String)session.getAttribute("session-uid");

out.println("Welcome "+ data+"!!");

%>

</body>

</html>

The control will be redirected to this page if the credentials entered by user are wrong.

failed.jsp

<html>

<head><title>Sign-in Failed Page</title>

</head>

<body>

<%

String data2=(String)session.getAttribute("session-uid");

out.println("Hi "+ data2+". Id/Password are wrong. Please try Again.");

%>

</body>

</html>

# OUT Implicit Object in JSP with examples

It’s an instance of **javax.servlet.jsp.JspWriter**. This allows us to access Servlet output stream. The output which needs to be sent to the client (browser) is passed through this object. In simple words out implicit object is used to write content to the client.

### Methods of OUT Implicit Object

void print()  
void println()  
void newLine()  
void clear()  
void clearBuffer()  
void flush()  
boolean isAutoFlush()  
int getBufferSize()  
int getRemaining()

Let’s see each of the out’s method in detail –

1. **void print()**: This method writes the value which has been passed to it. For e.g. the below statement would display a sentence Out Implicit Object in jSP – Book to the output screen (client browser).

out.print(“Out Implicit Object in jSP - Book”);

1. **void println()**: This method is similar to the print() method, the only difference between print and println is that the println() method adds a new line character at the end. Let’s have a look at the difference with the help of an example.  
   **print:**
2. out.print(“hi”);
3. out.print(" ");

out.print(“hello”);

output on browser: There will not be a new line between the outcomes of all 3 out.print statements.  
hi hello

**println:**

out.println(“hi”);

out.println(“hello”);

output on browser:  
hi  
hello

1. **void newLine()**: This method adds a new line to the output. Example –
2. out.print(“This will write content without a new line”);
3. out.newLine();

out.print(“I’m just an another print statement”);

**Output:**  
This will write content without a new line  
I’m just an another print statement

As you know print statement doesn’t add a new line. We have added a new line between two out.print statements using newLine() method.

1. **void clear()**: It clears the output buffer without even letting it write the buffer content to the client. This is how it can be called –

out.clear();

1. **void clearBuffer()**: This method is similar to the clear() method. The only difference between them is that when we invoke out.clear() on an already flushed buffer it throws an exception, however out.clearBuffer() doesn’t.
2. **void flush()** :  This method also clears the buffer just like clear() method but it forces it to write the content to the output before flushing it, which means whatever is there in buffer would be written to the client screen before clearing the buffer.
3. **boolean isAutoFlush()** :  It returns a Boolean value true/false. It is used to check whether the buffer is automatically flushed or not.
4. **int getBufferSize()**: This method returns the size of output buffer in bytes.
5. **int getRemaining()**: It returns the number of bytes remaining before hitting the buffer overflow condition.

### OUT Implicit Object Example

In this example we are using print and println methods of OUT for displaying few message to the client.

index.jsp

<HTML>

<HEAD>

<TITLE> OUT IMPLICIT OBJECT EXAMPLE </TITLE>

</HEAD>

<BODY>

<%

out.print( "print statement " );

out.println( "println" );

out.print("Another print statement");

%>

</BODY>

</HTML>

# Application Implicit Object in JSP with examples

Application implicit object is an instance of **javax.servlet.ServletContext**. It is basically used for getting initialization parameters and for sharing the attributes & their values across the entire JSP application, which means any attribute set by **application implicit object** would be available to all the JSP pages.

**Methods:**

* Object getAttribute(String attributeName)
* void setAttribute(String attributeName, Object object)
* void removeAttribute(String objectName)
* Enumeration getAttributeNames()
* String getInitParameter(String paramname)
* Enumeration getInitParameterNames()
* String getRealPath(String value)
* void log(String message)
* URL getResource(String value)
* InputStream getResourceAsStream(String path)
* String getServerInfo()
* String getMajorVersion()
* String getMinorVersion()

1. **Object getAttribute(String attributeName):** It returns the object stored in a given attribute name. For example the below statement would return the object stored in attribute “MyAttr”.

String s = (String)application.getAttribute("MyAttr");

1. **void setAttribute(String attributeName, Object object):** It sets the value of an attribute or in other words it stores an attribute and its value in application context, which is available to use across JSP application. Example –

application.setAttribute(“MyAttribute”, “This is the value of Attribute”);

The above statement would have stored the attribute and its value. What would be the value of ‘s’ if we use the below statement in any of the JSP page?

String s= (String) application.getAttribute(“MyAttribute”);

String s value would be “This is the value of Attribute” since we have set it using setAttribute method.

1. **void removeAttribute(String objectName):** This method is used for removing the given attribute from the application. For e.g. – It would remove the Attribute “MyAttr” from the application. If we try to get the value of a removed attribute using getAttribute method, it would return Null.

application.removeAttribute(“MyAttr”);

1. **Enumeration getAttributeNames():** This method returns the enumeration of all the attribute names stored in the application implicit object.

Enumeration e= application.getAttributeNames();

1. **String getInitParameter(String paramname):** It returns the value of Initialization parameter for a given parameter name. Example –web.xml
2. <web-app>
3. …
4. <context-param>
5. <param-name>parameter1</param-name>
6. <param-value>ValueOfParameter1</param-value>
7. </context-param>

</web-app>

Suppose above is my web.xml file

String s=application.getInitParameter(“parameter1”);

The value of s will be “ValueOfParameter1”. Still confused where did it come from? See the param-value tag in web.xml file.

1. **Enumeration getInitParameterNames():** It returns the enumeration of all the Initialization parameters.

Enumeration e= application.getinitParameterNames();

1. **String getRealPath(String value):** It converts a given path to an absolute path in the file system.

String abspath = application.getRealPath(“/index.html”);

The value of abspath would be a complete http URL based on the existing file system.

1. **void log(String message):** This method writes the given message to the JSP Engine’s (JSP container’s) default log file associated to the application.

application.log(“This is error 404 Page not found”);

The above call would write the message “This is error 404 Page not found” to the default log file.

1. **String getServerInfo():**This method returns the name and version of JSP container (JSP Engine).

application.getServerInfo();

**Application Implicit object Example**

A JSP page to capture number of hits using application. In this example we are counting the number of hits to a JSP page using application implicit object.

counter.jsp

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

<html>

<head>

<title>Application Implicit Object Example</title>

</head>

<body>

<%

//Comment: This would return null for the first time

Integer counter= (Integer)application.getAttribute("numberOfVisits");

if( counter ==null || counter == 0 ){

//Comment: For the very first Visitor

counter = 1;

}else{

//Comment: For Others

counter = counter+ 1;

}

application.setAttribute("numberOfVisits", counter);

%>

<h3>Total number of hits to this Page is: <%= counter%></h3>

</body>

</html>

# Config Implicit Object in JSP with examples

It is an instance of **javax.servlet.ServletConfig**. Config Implicit object is used for getting configuration information for a particular JSP page. Using application implicit object we can get application-wide initialization parameters, however using Config we can  get initialization parameters of an individual servlet mapping.

### Methods of Config Implicit Object

1. **String getInitParameter(String paramname)** – Same what we discussed in application implicit object tutorial.
2. **Enumeration getInitParameterNames()** – Returns enumeration of Initialization parameters.
3. **ServletContext getServletContext()** – This method returns a reference to the Servlet context.
4. **String getServletName()** – It returns the name of the servlet which we define in the web.xml file inside <servlet-name> tag.

**Config Implicit Object Example**

web.xml

Let’s say below is my **web.xml** file. I’m just defining servlet name and servlet mapping in it. Later, we would fetch few details from this file using config implicit object.

<web-app>

<servlet>

<servlet-name>BookServlet</servlet-name>

<jsp-file>/index.jsp</jsp-file>

</servlet>

<servlet-mapping>

<servlet-name>BookServlet</servlet-name>

<url-pattern>/index</url-pattern>

</servlet-mapping>

</web-app>

index.jsp

In this JSP page we are calling getServletName() method of config object for fetching the servlet name from web.xml file.

<html>

<head> <title> Config Implicit Object</title>

</head>

<body>

<%

String sname=config.getServletName();

out.print("Servlet Name is: "+sname);

%>

</body>

</html>

# pageContext Implicit Object in JSP with examples

It is an instance of **javax.servlet.jsp.PageContext**. Using this object you can find attribute, get attribute, set attribute and remove attribute at any of the below levels –

1. JSP Page – Scope: PAGE\_CONTEXT
2. HTTP Request – Scope: REQUEST\_CONTEXT
3. HTTP Session – Scope: SESSION\_CONTEXT
4. Application Level – Scope: APPLICATION\_CONTEX

### pageContext Implicit Object Example

index.html

Here we are simply asking user to enter login details.

<html>

<head>

<title> User Login Page – Enter details</title>

</head>

<body>

<form action="validation.jsp">

Enter User-Id: <input type="text" name="uid"><br>

Enter Password: <input type="text" name="upass"><br>

<input type="submit" value="Login">

</form>

</body>

</html>

validation.jsp

In this page we are storing user’s credentials using **pageContext** implicit object with the **session scope**, which means we will be able to access the details till the user’s session is active. We can also store the attribute using other scope parameters such as page, application and request.

<html>

<head> <title> Validation JSP Page</title>

</head>

<body>

<%

String id=request.getParameter("uid");

String pass=request.getParameter("upass");

out.println("hello "+id);

pageContext.setAttribute("UName", id, PageContext.SESSION\_SCOPE);

pageContext.setAttribute("UPassword", pass, PageContext.SESSION\_SCOPE);

%>

<a href="display.jsp">Click here to see what you have entered </a>

</body>

</html>

display.jsp

In this JSP page we are fetching the stored attributes using getAttribute method. The point to note here is that we have stored the attributes with session scope so we must need to specify scope as session in order to fetch those attribute’s value.

<html>

<head>

<title>Displaying User Details</title>

</head>

<body>

<%

String username= (String) pageContext.getAttribute("UName", PageContext.SESSION\_SCOPE);

String userpassword= (String) pageContext.getAttribute("UPassword", PageContext.SESSION\_SCOPE);

out.println("Hi "+username);

out.println("Your Password is: "+userpassword);

%>

</body>

</html>

### Methods of pageContext Implicit Object

1. **Object findAttribute (String AttributeName):** This method searches for the specified attribute in all four levels in the following order – Page, Request, Session and Application. It returns NULL when no attribute found at any of the level.
2. **Object getAttribute (String AttributeName, int Scope):**It looks for an attribute in the specified scope. This method is similar to findAttribute method; the only difference is that findAttribute looks in all the four levels in a sequential order while getAttribute looks in a specified scope. For e.g. – In the below statement the getAttribute method would search for the attribute “E.Balaguruswamy” in **Session scope (or Session level/layer).**If it finds the attribute it would assign it to Object obj else it would return Null.

Object obj = pageContext.getAttribute("E.Balaguruswamy ", PageContext.SESSION\_CONTEXT);

Similarly the method can be used for other scopes too –

Object obj = pageContext.getAttribute("E.Balaguruswamy", PageContext. REQUEST\_CONTEXT);

Object obj = pageContext.getAttribute("E.Balaguruswamy ", PageContext. PAGE\_CONTEXT);

Object obj = pageContext.getAttribute("E.Balaguruswamy ", PageContext. APPLICATION\_CONTEXT);

1. **void removeAttribute(String AttributeName, int Scope):**This method is used to remove an attribute from a given scope. For example – The below JSP statement would remove an Attribute “MyAttr” from page scope.

pageContext.removeAttribute(“MyAttr”, PageContext. PAGE\_CONTEXT);

1. **void setAttribute(String AttributeName, Object AttributeValue, int Scope):**It writes an attribute in a given scope. Example – Below statement would store an Attribute “mydata” in application scope with the value “This is my data”.

pageContext.setAttribute(“mydata”, “This is my data”, PageContext. APPLICATION\_CONTEXT);

Similarly this would create an attribute named attr1 in Request scope with value “Attr1 value”.

pageContext.setAttribute(“attr1”, “Attr1 value”, PageContext. REQUEST\_CONTEXT);

Exception Implicit Object in JSP with examples

We will discuss exception implicit object of JSP. It’s an instance of java.lang.Throwable and frequently used for exception handling in JSP. This object is only available for error pages, which means a JSP page should have isErrorPage to true in order to use exception implicit object. Let’s understand this with the help of below example –

### Exception implicit Object Example

In this example we are taking two integer inputs from user and then we are performing division between them. We have used exception implicit object to handle any kind of exception in the below example.

index.html

<html>

<head>

<title>Enter two Integers for Division</title>

</head>

<body>

<form action="division.jsp">

Input First Integer:<input type="text" name="firstnum" />

Input Second Integer:<input type="text" name="secondnum" />

<input type="submit" value="Get Results"/>

</form>

</body>

</html>

Here we have specified exception.jsp as errorPage which means if any exception occurs in this JSP page, the control will immediately transferred to the exception.jsp JSP page. Note: We have used **errorPage attribute of Page Directive** to specify the exception handling JSP page (<%@ page errorPage=”exception.jsp” %>).

division.jsp

<%@ page errorPage="exception.jsp" %>

<%

String num1=request.getParameter("firstnum");

String num2=request.getParameter("secondnum");

int v1= Integer.parseInt(num1);

int v2= Integer.parseInt(num2);

int res= v1/v2;

out.print("Output is: "+ res);

%>

In the below JSP page we have set **isErrorPage to true** which is also an attribute of Page directive, used for making a page eligible for exception handling. Since this page is defined as a exception page in division.jsp, in case of any exception condition this page will be invoked. Here we are displaying the error message to the user using **exception implicit object**.

exception.jsp

<%@ page isErrorPage="true" %>

Got this Exception: <%= exception %>

Please correct the input data.

# JSP Expression Language (EL)

Expression language (EL) has been introduced in JSP 2.0. The main purpose of it to simplify the process of accessing data from [bean](https://beginnersbook.com/2013/11/jsp-usebean-setproperty-getproperty-action-tags/) properties and from [implicit objects](https://beginnersbook.com/2013/11/jsp-implicit-objects/). EL includes arithmetic, relational and logical operators too.

**Synatx of EL:**

${expression}

whatever present inside braces gets evaluated at runtime and being sent to the output stream.

**Example 1: Expression language evaluates the expressions**

In this example we are evaluating the expressions with the help of EL.

<html>

<head>

<title>Expression language example1</title>

</head>

<body>

${1<2}

${1+2+3}

</body>

</html>

**Example 2: Value fetch using param variable of expression language**

In this example we are prompting user to enter name and roll number. On the other JSP page we are fetching the entered details using param variable of EL.

index.jsp

<html>

<head>

<title>Expression language example2</title>

</head>

<body>

<form action="display.jsp">

Student Name: <input type="text" name="stuname" /><br>

Student RollNum:<input type="text" name="rollno" /><br>

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

</form>

</body>

</html>

display.jsp

<html>

<head>

<title>Display Page</title>

</head>

<body>

Student name is ${ param.stuname } <br>

Student Roll No is ${ param.rollno }

</body>

</html>

**Example 3: Getting values from application object.**

In this example we have set the attributes using application implicit object and on the display page we have got those attributes using **applicationScope** of Expression language.

index.jsp

<html>

<head>

<title>EL example3</title>

</head>

<body>

<%

application.setAttribute("author", "Chaitanya");

application.setAttribute("Site", "Facebook.com");

%>

<a href="display.jsp">Click</a>

</body>

</html>

display.jsp

<html>

<head>

<title>Display Page</title>

</head>

<body>

${applicationScope.author}<br>

${applicationScope.Site}

</body>

</html>

**EL predefined variables:**

Similar to implicit objects in JSP we have predefined variables in EL. In the above examples we have used param and applicationScope, they are also the part of these variables.

**pageScope**: It helps in getting the attribute stored in Page scope.  
**pageContext**: Same as JSP PageContext object.  
**sessionScope**: Fetches attributes from session scope, set by [session object](https://beginnersbook.com/2013/11/jsp-implicit-object-session-with-examples/).  
**requestScope**: It used for getting the attributes from request scope. The attribute which are set by [request implicit object](https://beginnersbook.com/2013/11/jsp-implicit-object-request-with-examples/).  
**param**: Similar to **ServletRequest.getParameter**. See Example 2.  
**applicationScope**: Used for getting [Applicaton](https://beginnersbook.com/2013/11/jsp-implicit-object-application-with-examples/) level attributes. Same what we see in Example 3.  
**header**: It helps in getting HTTP request headers as Strings.  
**headerValues**: Used for fetching all the HTTP request headers.  
**initParam**: It links to context initialization parameters.  
**paramValues**: Same as ServletRequest.getParmeterValues.  
**cookie**: It maps to Cookie object.

# JSP Custom tags with example

User-defined tags are known as **custom tags**. In this tutorial we will see how to **create a custom tag**and use it in JSP.

To create a custom tag we need three things:  
**1) Tag handler class**: In this class we specify what our custom tag will do when it is used in a JSP page.  
**2) TLD file**: Tag descriptor file where we will specify our tag name, tag handler class and tag attributes.  
**3) JSP page**: A JSP page where we will be using our custom tag.

**Example:**  
In the below example we are creating a custom tag MyMsg which will display the message “This is my own custom tag” when used in a JSP page.

**Tag handler class:**  
A tag handler class should implement Tag/IterationTag/ BodyTag interface or it can also extend TagSupport/BodyTagSupport/SimpleTagSupport class. All the classes that support custom tags are present inside javax.servlet.jsp.tagext. In the below we are extending the class SimpleTagSupport.

Details.java

package facebook.com;

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

import javax.servlet.jsp.\*;

import java.io.\*;

public class Details extends SimpleTagSupport {

public void doTag() throws JspException, IOException {

/\*This is just to display a message, when

\* we will use our custom tag. This message

\* would be displayed

\*/

JspWriter out = getJspContext().getOut();

out.println("This is my own custom tag");

}

}

**TLD File**  
This file should present at the location: Project Name/WebContent/WEB-INF/ and it should have a **.tld** extension.

**Note:**  
<name> tag: custom tag name. In this example we have given it as MyMsg  
<tag-class> tag: Fully qualified class name. Our tag handler class Details.java is in package so we have given the value.

message.tld

<taglib>

<tlib-version>1.0</tlib-version>

<jsp-version>2.0</jsp-version>

<short-name>My Custom Tag</short-name>

<tag>

<name>MyMsg</name>

<tag-class>facebook.com.Details</tag-class>

<body-content>empty</body-content>

</tag>

</taglib>

**Using custom tag in JSP:**

Above we have created a custom tag named MyMsg. Here we will be using it.

**Note**: taglib directive should have the TLD file path in uri field. Above we have created the message.tld file so we have given the path of that file.  
Choose any prefix and specify it in taglib directive’s prefix field. Here we have specified it as myprefix.  
Custom tag is called like this: <prefix:tagName/>. Our prefix is myprefix and tag name is MyMsg so we have called it as <myprefix:MyMsg/> in the below JSP page.

<%@ taglib prefix="myprefix" uri="WEB-INF/message.tld"%>

<html>

<head>

<title>Custom Tags in JSP Example</title>

</head>

<body>

<myprefix:MyMsg/>

</body>

</html>

**Output:**

This is my own custom tag

# How to access body of Custom tags in JSP

Now we will see how to access the body of custom tag. For e.g. If our custom tag is xyz then we would learn to access the content between <prefix: xyz> and </prefix:xyz>

<prefix: xyz>

**Body of custom tag**: **This is what we will access in the below example**

</prefix:xyz>

**Example:**  
In this example or custom tag will append a String to its own body and will display the result.

**Tag handler class:** Details.java

package facebook.com;

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

import javax.servlet.jsp.\*;

import java.io.\*;

public class Details extends SimpleTagSupport {

//StringWriter object

StringWriter sw = new StringWriter();

public void doTag() throws JspException, IOException

{

getJspBody().invoke(sw);

JspWriter out = getJspContext().getOut();

out.println(sw.toString()+"Appended Custom Tag Message");

}

}

**TLD file:** message.tld  
Remember to have this file in WEB-INF folder.

<taglib>

<tlib-version>1.0</tlib-version>

<jsp-version>2.0</jsp-version>

<short-name>My Custom Tag: MyMsg</short-name>

<tag>

<name>**MyMsg**</name>

<tag-class>**facebook.com.Details**</tag-class>

<body-content>**scriptless**</body-content>

</tag>

</taglib>

**JSP Page:**index.jsp

<%@ taglib prefix="**myprefix**" uri="**WEB-INF/message.tld**"%>

<html>

<head>

<title>Accessing Custom Tag Body Example</title>

</head>

<body>

<**myprefix**:**MyMsg**>

**Test String**

</**myprefix**:**MyMsg**>

</body>

</html>