# Java Server Pages (JSP)

It is a technology used to create dynamic webpage. Java Server Pages is a Java-based technology that simplifies the process of developing dynamic web sites. With JSP, web designers and developers can quickly incorporate dynamic elements into web pages using embedded java. JSP files contain traditional HTML along with embedded code that allows the page designer to access data from java code running on the server.

Based on the Java programming language, Java Server Pages offers proven portability, open standards, and mature re-usable component model .The Java Server Pages architecture enables the separation of content generation from content presentation. This separation not eases maintenance headaches; it also allows web team members to focus on their areas of expertise. Now, web page designer can concentrate on layout, and web application designers on programming, with minimal concern about impacting each other's work.

To make creation of dynamic content easier, Sun introduced Java Server Pages (JSP). While writing served developer require a pretty extensive knowledge of Java, a person who is new to the java can learn some pretty neat tricks in JSP in a snap. JSP represents an attractive alternative of Microsoft's ASP(Active Server Pages). JSP provides the HTML designer with a way to access data and business logic stored inside Java objects without having to master the complexities of Java application development.

When a user will access a servlet ,then the runtime environment or web container executes the servlet in servlet engine ,but when a jsp is accessed by the user ,then the web container extracts the java code available within the jsp file and creates a servlet, then the web container compiles(by jasper) & executes(by catalina) the created servlet in the servlet container. once the servlet is executed then the web server collects the data/output of the servlet and embed the o/p of the servlet within html tag of the jsp file to send the response back to the user.

The web server will not throw away the servlet created during the first request of the user, rather the web server will store the servlet in cache memory to process the request of subsequent users. In case of servlet, the html code is written within the java syntax i.e. within println() but in case of jsp, the java code is written within scriptlet <% %> tag or we can say the java code will be written along with html tag.

Although servlets are powerful web components, they are not ideal technology available to us to build presentation elements. This is because **:--**

1) Amending the look and feel of the system involves recompiling the Servlet classes.
2) The presence of HTML within the Servlet tightly couples the presentation and the content, which blurs the role of presentation and providing content.

Lots of HTML code within the Servlet classes makes them difficult to maintain, Java Server pages addresses these problems. JSP are text files similar to HTML files but have extra tags that allow us to **:**

- Interact with dynamic content.
- Include content from other web application resources.
- Forward the response to other web application resources.

**Benefits of JSP:**

- As a Java-based Technology, it enjoys all the advantages that the java language provides with respect to development and deployment.
- As an Object Based Language with a strong typing encapsulation, exception handling, and automatic memory management, use of Java leads to increase programmer productivity and more robust code.
- Because compiled Java byte code is portable across all platforms that support JVM, use of JSP does not lock you into using specific hardware platform, operating system, or server software.
- JSP is vendor- neutral, developers and system architects can select best-of-breed solutions at all stages of JSP deployment.
- JSP can readily take advantages of the other entire standard Java API's including those for cross-platform database access, directory services, distributed computing and cryptography.
- Provides support for a re-usable component such as Java-Beans Technology.
- It provides the easy way to develop the code as compared to Servlet Technology of the Java.

Quite a lot of things happen behind the scene when a JSP page is deployed in a web container and is first served to a client request.

**Features of JSP**

**Portability:** Java Server Pages files can be run on any web server or web-enabled application server that provides support for them. Dubbed the JSP engine, this support involves recognition, translation, and management of the Java Server Page lifecycle and its interaction components.

**Components:** It was mentioned earlier that the Java Server Pages architecture can include reusable Java components. The architecture also allows for the embedding of a scripting language directly into the Java Server Pages file. The components current supported include Java Beans and Servlets.

**Processing:** A Java Server Pages file is essentially an HTML document with JSP scripting or tags. The Java Server Pages file has a JSP extension to the server as a Java Server Pages file. Before the page is served, the Java Server Pages syntax is parsed and processed into a Servlet on the server side. The Servlet that is generated outputs real content in straight HTML for responding to the client.

**Access Models:** A Java Server Pages file may be accessed in at least two different ways. A client's request comes directly into a Java Server Page. In this scenario, suppose the page accesses reusable Java Bean components that perform particular well-defined computations like accessing a database. The result of the Beans computations, called result sets is stored within the Bean as properties. The page uses such Beans to generate dynamic content and present it back to the client. In both of the above cases, the page could also contain any valid Java code. Java Server Pages architecture encourages separation of content from presentation.

**LIFE CYCLE OF JSP:** A JSP life cycle can be defined as the entire process from its creation till the destruction which is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet. The following are the paths followed by a JSP

 Compilation  Initialization  Execution  Cleanup

The four major phases of JSP life cycle are very similar to Servlet Life Cycle and they are as follows:

**JSP Compilation/ Translation Phase:** In this phase the JSP page is transformed into a Java Servlet and then compiled. This phase occurs only once for each JSP page before the JSP page is served. When a browser asks for a JSP, the web server(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 recompiles the page. The compilation process involves three steps:

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

**[ Implementation class:** The translation phase results in a delay when a JSP page is requested for the first time. To avoid this delay, JSP pages can be precompiled before they are deployed using tools that perform the translation phase when the server starts up. **]**

**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 it is s similar to init() of Servlet, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

**JSP Execution:** This phase (also known as the Request Processing phase) is executed each time the JSP page is served by the web container. Request for the JSP page result in the execution of the JSP page implementation class. 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 once per a request and 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 ie. GET, POST, DELETE etc.

**JSP Cleanup:** The destruction phase of the JSP life cycle begins when a JSP is being removed from the web 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.
}
```

## CHARACTERSTICS OF JSP:-

1) **INTERPRETER: -** A JSP file will never get compiled by the user, rather it is compiled & interpreted by the web server( javac is the compiler for servlet, jasper is the compiler of jsp(i.e. servlet created from jsp), catalina is the interpreter for servlet & jsp)
2) **OBJECT BASE: -** JSP has some predefined object like request, response, exception, application, session, out etc by which we can perform various operation. JSP doesn't support Inheritance, Polymorphism. Hence JSP is Object Based & servlet is object oriented.
   a) **response**: This Object behaves like HttpServletResponse interface to provide response to the user.
   b) **reques**t: This Object behaves like HttpServletRequest interface by which we can get data of the user.
   c) **exception**: It is an Object of java.Lang.Throwable call by which we can handle exception in JSP.
   d) **application**: It is an Object which is similar to ServletContext interface by which we can create application level variable.
   e) **session**: It is an Object of HttpSession class which is used to create user session of individual user.
   f) **out**: It is an object of PrintWriter class by which we can invoke println().

3) **Tag Based**: JSP supports some predefined tags by which we can write Syntaxes. JSP also supports user defined tags. Hence JSP able to achieve code reuse without the inheritance.
4) **User Friendly**: Since the HTML code & tag will remain outside of JSP tag. Hence it is easy to design the HTML file by using IDE such as Netbeans, Eclipse, DreamWeaver etc.

| SERVLET | JSP |
|---|---|
| It is complied by user & interpreted in server. | It is interpreted by server |
| It is Object Oriented. | It is object based. |
| HTML code appears within JAVA code. | JAVA code will appear within HTML file. |
| It doesn't support user defined tags. | It is support user defined tags. |

## CREATION PROCESS OF JSP

1. Create JSP file by providing **.JSP** extension.
2. Write java code within **scriptlet** tag as given bellow :-
   ```
   <%
   Java  code  goes  here
   %>
   ```
3. The JSP file should be stored directly in **Context Folder** i.e. **root directory** of the website.
4. Start the server & deploy/host the website. Access the JSP file by providing file name in the URL along with URL of website.

## STEPS IN THE EXECUTION OF A JSP APPLICATION:

1. The client sends a request to the web server for a JSP file by giving the name of the JSP file within the form tag of a HTML page or in url of href attribute of anchor tag.

2. This request is transferred to the JavaWebServer. At the server side JavaWebServer receives the request and if it is a request for a jsp file server gives this request to the JSP engine.

3. JSP engine is program which can understands the tags of the jsp and then it converts those tags into a Servlet program and it is stored at the server side. This Servlet is loaded in the memory and then it is executed and the result is given back to the JavaWebServer, and then the result is given back to the client.

## CONTENTS OF JSP

1. **DIRECTIVE TAGS:** This tag is used to import any package or file to a JSP file. Generally this tag must be given at the beginning of the JSP.
   **Syntax:**
   <%@ directive_name attribute=value %>
   **Example**
   <%@ page import=”java.util.*, java.sql.*” %>
   Page: - Directive Name
   Import: - Attribute Name
   <%@ include file=”login.jsp” %> - - > It’ll include the content/code of login.jsp in another jsp.

| **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 |

2. **DECLARATION SECTION/TAG:** This section allows us to declare variables & objects which will be used in a JSP. This tag must be written before we can provide any response given to the client. This tag will never appear within any other tag. It must be given before any scriptlet tag.
   **Syntax:**
   <%! variables & object declaration %>
   **Example:**
   <%!
   String s1,s2=null;
   int a,b=0;
   %>
3. **CLIENT END SCRIPTING TAG:** This section doesn’t belongs to JSP but JSP allows us to provide client side validation by using javascript or vbscript.
   Example:
   <script language=”javascript”>
   Scripting language/validation code goes here
   </script>
4. **HTML:** Java allows us to provide HTML tag written JSP.

5. **JSP TAG/ACTION TAG:** Java has given some predefined tag in JSP to use bean class, custom tag/user defined tag in JSP. As well as Java allows us to use user defined tag/custom tag within a JSP file. This tag will never appear within any other tag.

| **Syntax** | **Purpose** |
|---|---|
| jsp:include | Includes a file at the time the page is requested |
| 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. |

6. **SCRIPTLET TAG:** This tag allows us to write Java syntax within a JSP file. This tag will never appear within any other tag, but this tag may appear several times in a JSP file.
Syntax:
```
<%
Java codes
%>
```
7. **EXPRESSION TAG:** This tag is used to print the result of an expression or a variable directly within JSP file. This tag may appear within any other tag. We can say this tag is a separate mechanism to print value of variable or expression without println(). This tag will never appear within scriptlet tag. This tag should nota contain semicolon otherwise compilation error will occur.
Example:
```
<%!   Int x=12, y=13;   %>
Addition Result is : <%=    x+y      %>
```

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

| | |
|---|---|
| <%-- This is JSP comment --%> | It is ignored by JSP engine. |
| <!-- This is html comment --> | It is ignored by browser. |

9. JSP IMPLICIT OBJECTS : JSP supports nine automatically defined variables, which are also called implicit objects.

| **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**. It is an object of javax.servlet.jsp.**PageContext** |
| page | This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class. It's an object of javax.servlet.jsp.**HttpJspPage** |
| exception | The **Exception** object allows the exception data to be accessed by designated JSP. |

| **Example(First.jsp)** | **Second.jsp** |
|---|---|
| `<html><body bgcolor="pink">`<br>`<h1>  Welcome to First jsp <br>`<br>`<%`<br>`int x=12, y=23;`<br>`int z= x+y;`<br>`out.println("sum of "+x+" and "+y+"="+z);`<br>`%>`<br>`</h1></body></html>` | `<html><body bgcolor="cyan">`<br>`<h1>  Welcome to Second jsp <br><table border=2>`<br>`<%  for(int i=0; i<5; i++)  {`<br>`out.println("<tr>");`<br>`for(int j=0; j<10; j++)  {`<br>`out.println("<td>  kalia</td>");`<br>`}`<br>`out.println("</tr>");`<br>`}    %>`<br>`</h1></body></html>` |

**Submitting Request JSP to itself :** If in a jsp file if we have not given the action attribute of the form tag then the data of the jsp page will be submitted to itself.

**Itself.jsp :**

```
<html><body  bgcolor="cyan">
<%!  String na, pw; %>
<%
  try{       na=request.getparameter("t1");
              pw=request.getparameter("t2");
  }catch(Exception e){ }
      if( na==null || na.equals("")){
%>
            <form method="get"><h1>
            user name<input type="text" name="t1"><br>
            password<input  type="password" name="t2"><br>
            <input type="submit" value="submit" /><input type="reset" value="reset" /></h1></form>
<%   } else  {
            out.printin("Welcome"+na+"ur pass"+pw);
        }
 %>
</body></html>
```

## Using DataBase in JSP with life cycle methods :

**Store as stud.html:**

```
<html>
```

```
	<head><title> Student details </title></head>
	<body><form method="get" action="getstud.jsp"><table border='1' align="center">
				<tr>
					<td>Student Roll Number</td>
					<td><input type="text" name="rno" value=""></td>
				</tr>
				<tr>
					<td><input type="submit" name="button" value="Submit"></td>
				</tr>
	</table></form></body>
</html>
```

**<u>Store it as getstud.jsp :</u>**

```
<%--
	JSP page design steps
		1. In jspInit() method
			create   Connection, Statement, PreparedStatement objects
			by reading driver information from web.xml file

			Note: In jsp based project web.xml is mandatory only if
				1. jsp file want to be configured as welcome file
				2. to read jsp initialization parameters

		2. In _jspService() method [Scriptlet tag] read student information from DB and display it to end user
with HTML table component

		3. In jspDestroy() method release all JDBC objects.

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

<%!	Connection	con;
	Statement	stmt;
	ResultSet	rs;

	public void jspInit(){
		try{
/*  reading jsp intialization values from web.xml file, means from ServletConfig object. retrieving ServletConfig
object  */
			ServletConfig	config	= getServletConfig();
			ServletContext	ctxt		= config.getServletContext();

			String driver	= ctxt.getInitParameter("driver");
			String url		= ctxt.getInitParameter("url");
			String usn		= config.getInitParameter("usn");
			String pwd		= config.getInitParameter("pwd");

			//preparing JDBC objects
```

```
                Class.forName(driver);
                con             = DriverManager.getConnection(url, usn, pwd);
                stmt            = con.createStatement();
            }catch(Exception e){    e.printStackTrace();            }
        }
        public void jspDestroy(){
            //releasing JDBC objects
            try{
                if(stmt != null)  {
                    stmt.close();
                    stmt = null;
                }

                if(con != null){
                    con.close();
                    con = null;
                }
            }catch(Exception e){
                e.printStackTrace();
            }
        }
%>

<%
try{
            //reading sno from html form
            String rol = request.getParameter("rno").trim();

            String query;
            //checking user decision
            if(rol == null || rol.equals("")){
                //user want to list all students details
                query = "SELECT * FROM stud";
            }else    {
                //user want to retrieve only given student information
                query = "SELECT * FROM stud WHERE roll ="+ rol;
            }
            rs = stmt.executeQuery(query);
            //preparing presentation logic
        %>

        <html> <head> <title> student details </title></head>
        <body> <h1> Here the student roll number <%= rol %> details
        <table border="1" align="center">
<tr> <th>Name</th> <th>Roll</th> <th>Sem</th> <th>Mark </th> </tr>
        <%      while (rs.next())   {  %>
            <tr> <td><%= rs.getString(1) %></td>      <td><%= rs.getInt(2) %></td>
```

```
                    <td><%= rs.getString(3) %></td>  <td><%= rs.getFloat(4) %></td>    </tr>
          <%      }    %>
          </table></form></body>
          </html>
         <%
rs.close();
          }catch(Exception e){   e.printStackTrace();      }
%>
```

**web.xml :**

```
<context-param>
  <param-name>driver</param-name>
  <param-value>com.mysql.jdbc.Driver</param-value>
 </context-param>

 <context-param>
  <param-name>url</param-name>
  <param-value>jdbc:mysql://localhost/college</param-value>
 </context-param>

 <servlet>
  <servlet-name>select</servlet-name>
  <jsp-file>/getstud.jsp</jsp-file>

  <init-param>
   <param-name>usn</param-name>
   <param-value>root</param-value>
  </init-param>

  <init-param>
   <param-name>pwd</param-name>
   <param-value></param-value>
  </init-param>
 </servlet>

 <servlet-mapping>
<servlet-name>select</servlet-name>
<url-pattern>/getstud.jsp</url-pattern>
</servlet-mapping>

</web-app>
```

**USER TRACKING :** It can be achieved by followings: -

1. Url rewriting
2. Hidden field
3. Cookie
4. Session

1) **URL rewriting** **(Store it as URLDemo.html )**

```
<html><body bgcolor="yellow">
<form  action="UDemo1.jsp"  method ="get"><h1>
User Name <input type="text" name="t1" value="ur  name"><br>
Password <input type="password" name="p1"><br>
<input  type="submit" value="Submit"><input  type="reset" value="Reset">
</h1></form></body></html>
```

| **UDemo1.jsp** | **UDemo2.jsp :** |
|---|---|
| `<html><body bgcolor=cyan>`<br>`<%! String s1, s2;  %>`<br>`<h1>Welcome to URL rewriting<br>`<br>` <%      s1 = request.getParameter("t1");`<br>`           s2 = request.getParameter("p1");`<br>`%>`<br>`<a href="UDemo2.jsp?a1=<%= s1 %> &a2=<%= s2 %>"> Click Me</a>`<br>`</h1></body></html>` | `<html><body bgcolor=cyan>`<br>`<%! String s1, s2;  %>`<br>` <%`<br>`   s1 = request.getParameter("a1");`<br>`   s2 = request.getParameter("a2");`<br>`%>`<br>`<h1>Welcome <%= s1 %> ur password is <%= s2 %>`<br>`</h1></body></html>` |

**<u>2) Hidden field:</u> (Store it as <u>HidDemo.html )</u>**

```
<html><body bgcolor="yellow">
<form  action="Hid1.jsp"  method ="post"><h1>
User Name <input type="text" name="t1" value="ur  name"><br>
<input  type="submit" value="Submit">
</h1></form></body></html>
```

| **Hid1.jsp** | **Hid2.jsp** |
|---|---|
| `<html><body bgcolor=cyan>`<br>`<%! String s1;  %>`<br>`<h1>Welcome to Hiiden Field<br>`<br>` <%    s1 = request.getParameter("t1");       %>`<br>` <form action ="Hid2.jsp"  method="post">`<br>`<input type="hidden" name="t1" value="<%= s1 %>">`<br>`Enter ur phone no<input type="text" name="t2">`<br>`<input type="submit" value="submit">`<br>`</form></h1></body></html>` | `<html><body bgcolor=cyan>`<br>`<%!    String s1, s2;  %>`<br>` <%     s1 = request.getParameter("t1");`<br>`           s2 = request.getParameter("t2");`<br>`%>`<br>`<h1>Welcome <%= s1 %> ur phno is <%= s2 %>`<br>`</h1></body></html>` |

**<u>3) Cookie :</u>      <u>(Store it as CookDemo.html )</u>**

```
<html><body bgcolor="magenta">
<form  action="Cookie1.jsp"  method ="get"><h1>
User Name <input type="text" name="t1" ><br>
password <input type="password" name="t2" ><br>
<input  type="submit" value="Submit">
</form></body></html>
```

| **Cookie1.jsp** | **Cookie2.jsp** |
|---|---|

| | |
|---|---|
| `<html><body bgcolor="cyan">`<br>`<%    String s1= request.getParameter("t1");`<br>`        String s2= request.getParameter("t2");`<br>`        Cookie  c1=new  Cookie("name", s1);`<br>`        Cookie  c2=new  Cookie("pwd", s2);`<br>`        response.addCookie(c1);`<br>`        response.addCookie(c2);`<br>`%>`<br>`<h1>Welcome to Cookie <a  href="Cookie2.jsp">`<br>`Testing ur Cookie</a></h1></body></html>` | `<html><body bgcolor="pink">Cookie example <br>`<br>`<%    Cookie ck[]= request.getCookies();`<br>`        for(int i=0; i< ck.length;i++) {`<br>`%>`<br>`            Variable name :  <%= ck[i].getName() %>`<br>`value is : <%= ck[i].getValue() %><br>`<br>`<%     }  %>`<br>`</h1></body></html>` |

**4) Session:** **(Store it as SessDemo.html)**

```
<html><body bgcolor="yellow" >
<form   action="Sess1.jsp"><h1>
Enter ur name  &nbsp;&nbsp;<input type="text" name="t1"> <br>
Enter password  &nbsp;&nbsp;<input type="password" name="t2"><br>
<input type="submit" value="Submit"></h1></form></body></html>
```

| **Sess1.jsp** | **Sess2.jsp** |
|---|---|
| `<html><body bgcolor=red>`<br>`<h1>Welcome to session<br>`<br>`<%!  String  s1, s2;  %>`<br>`<%    s1=request.getParameter("t1");`<br>`        s2=request.getParameter("t2");`<br>`         session.setAttribute("uname",s1);`<br>`        session.setAttribute("pwd", s2);`<br>`%>`<br>`<a  href="Sess2.jsp">Click</a>`<br>`</h1></body></html>` | `<html><body bgcolor=pink>`<br>`<h1>The value of ur session is :2nd JSP<br>`<br>`<%!  String p1, p2;  %>`<br>`<%`<br>` p1=(String) session.getAttribute("uname");`<br>` p2=(String) session.getAttribute("pwd");`<br>`%>`<br>`user name : <%= p1 %> <br>`<br>`password  : <%= p2 %>`<br>`</h1></body></html` |

**INCLUDING AND FORWARDING THE REQUEST:** When HTML will send the reqeust to first JSP, then the first JSP will process the request and provide the response back to the browser. But the users request will not be available to 2nd / 3rd JSP. The request forwarding mechanism allows us to propagate the same request of the user to 1st, 2nd, 3rd JSP and so on. The last JSP will provide the response to the user, other JSP will remain invisible to the user.

The request forwarding mechanism in servlet provide extra information to the other servlet. This mechanism is also available in accessing a JSP from servlet in JSP chapter because it will forward the same request and response of the user to the other survlet. But in case of JSP we can also add extra information from one JSP to other alongwith the users request. The request forwarding mechanism in JSP can be achieved by <jsp: forward> as givewn below

```
<jsp: forward  Page = “For2. JSP”>
<jsp: param name= “name”  value = “<%= s2 %>”/>
</jsp: forward>
```

The <jsp: forward> is a predefined JSP tag and it will hold the address of next JSP file in page attribute to which the users request will be forwarded. Let us say, we have written the above code in For1. jsp, the request available to For1. jsp will be forwarded to For2. Jsp, as well as For1. jsp may process the users reqeust on the data base or can implement any logic on the users request, then For1. jsp may forward the users request to For2. jsp. The For1. jsp is also able to provide extra information or value to For2.jsp by using jsp: param subtag of < jsp:forward. The for2.jsp will get the users request by using getAttribute() as well as the value given in <param > by providing the name given in <param> in getAttribute().

If we do not provide the jsp:param subtag then we can write the<jsp:forward> in one line as <jsp:forward page = “for2.jsp” />

Similarly, the <jsp: include >allows us to include content of another jsp./HTML file into current jsp file.

<jsp:include page = “login.html”/>

**Example (Forward,html)**

```
<html><body bgcolor="yellow"> <h1>
<form  action="Forward1.jsp"  method ="get">
User Name <input type="text" name="t1" value="ur  name"><br>
Password <input type="password" name="t2"><br>
<input  type="submit" value="Submit">
<input  type="reset" value="Reset"></h1>
</form></body></html>
```

**Forward1.jsp :**

```
<%@ page import="java.sql.*"  %>
<html><body bgcolor="yellow"><h1>
<%!   String na, pw, ud, ty;     %>
<%
      na = request.getParameter("t1");
      pw = request.getParameter("t2");
 try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection  con = DriverManager.getConnection("jdbc:odbc:gaura", "root","");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from userlog where uname='"+na+"'  and  pwd='"+pw+"'");
    if(rs.next()){
        ud = rs.getString("uid");
        ty = rs.getString("type");
%>
     <jsp:forward page="Forward2.jsp">
     <jsp:param name="uid" value="<%= ud %>" />
     <jsp:param name="type" value="<%= ty %>" />
     </jsp:forward>
<%
  }else{
%>
      <jsp:include page="Login.html" />
      Invalid username/password, try again
<%
  }
    con.close();
}catch(Exception e){ }
 %>
</h1></body></html>
```

**Forward2.jsp :**

```
<html><body bgcolor="yellow"><h1>
<%!   String ename, pass, empno, etype;     %>
<%
       ename = request.getParameter("t1");
       pass= request.getParameter("t2");
       empno = request.getParameter("uid");
       etype = request.getParameter("type");
%>
Welcome <%= ename %> <br>
Ur password <%= pass %><br>
Ur user id  <%= empno %><br>
Ur type <%= etype %><br>
</h1></body></html>
```

**STATIC INCLUDE :** In this mechanism the content of one jsp file is included within another jsp.file. In case of static include the content of one jsp file will be included within another jsp before the jsp is converted into servlet. But in case <jsp:include> the result/response/output of one jsp file is included within an other jsp file.

The static include helps us to avoid the duplicacy of a code, so that we can use the same code in several jsp. For example if we create a jsp called dbcon.jsp which will contain database connection, then in any jsp when we go for database interaction we should not write Class.forName(), Connection con, Statement st, because we can reuse these lines available in dbcon.jsp in other jsp. But if we like to use same design in all pages like header.jsp, footer.jsp etc. Which will be included in any jsp page of the website so that the designing of header and footer will appear same in all pages, then we can use <jsp:include>. Further, if we want to change anything in header/footer, then only the header.jsp/footer.jsp will be changed. No need to change all the web pages.

**EXCEPTION HANDLING IN JSP :** When an exception occurs in a jsp, then the info of the exception is directly displayd on the browser.

This mechanism redirects the exception/error of a jsp file to an error page(error page is also another jsp). When exception occurs in a jsp then control will be transferred to error page for handling the exception as well asto avoid the exception stack/info on the browser.

The jsp file in which the exception may occur must provide the jsp file name which will handel the exception in **errorPage** attribute of pag directive.

Ex : <%@ page errorPage="B.jsp" /> , say we have written this statement in A.jsp & when exception will occur in A.jsp then control will be transferred to B.jsp file.

To implements this mechanism we can creae one jsp file(say B.jsp) to handel the exception(papularly called error page) for entire website. This page must provide "true" to **isErrorPage** attribute of page directive, this page must use predefined **exception** object to deal with exception,

Ex : <%@ page isErrorPage="true" /> , say we have written this statement in B.jsp, hence this file handel the exception.

**TestError.html :**

```
<html><body bgcolor="dhala"><h1>
<form   action="ErrFind.jsp">
Enter a number<input type="text" name="t1">
<input type="submit" value="submit">
</form></h1></body></html>
```

| **ErrFind.jsp :** | **error.jsp** |
|---|---|
| `<%@ page errorPage="error.jsp" %>`<br>`<html><body bgcolor="asdfgh"><h1>`<br>`Welcome to calculation page<br>`<br>`<%!      String s1;      int x;      %>`<br>`<%`<br>`  s1 = request.getParameter("t1");`<br>`  x = Integer.parseInt(s1);`<br>`  if(x%2 ==0) {`<br>`%>`<br>` <%= x %> is even number.`<br>`<%  } else {  %>`<br>` <%= x %> is odd number.`<br>`<%  }  %>`<br>`</h1></body></html>` | `<%@ page isErrorPage="true" %>`<br>`<html><body bgcolor="iyasgh"><h1>`<br>`Error Handling in jsp<br>`<br>`<%`<br>`   out.println(exception.getMessage());`<br>` %>`<br>`Error in value <a href="TestError.html">try`<br>`again</a>`<br>`</h1></body></html>` |

**JAVA BEAN :** Java Bean is a reusuable component (bean is a simple java class) Java bean provides a mechanism to create development component. The components are part of the system / software which is used to perform a well defined task. A component may be a visible component / it may be an invisible component.

A visible component works with the user interface of the s/w whereas an invisible component will work behind the user interface of the s/w. A normal java class with a set of convention for its propertis and behaviour (data member and member function) is known as javan bean. The properties of the bean class must be accessed / modified by the methods of Bean class.

**Contents of Java Bean**

a. Properties (data member)

b. Method
c. Event

**a) Properties:** The variables declared within java class is called properties of the Bean class .The properties represents state of the java bean.A Bean class has 5 types of properties as given below:

i. Simple Property
ii. Indexed Property
iii. Boolean Property
iv. Bound Property
v. Constraint Property

**i. Simple Property :** If the data member of Bean class doesnt belong to boolean datatype of an array,then it is called Simple Property.

E.g int x; String name;

**ii. Indexed Property :** If the data member of Bean class is an array which can be modified by using index/counter,then it is called Indexed property .

E.g int a[];

**iii. Boolean Property :** If the data member of Bean class belongs to boolean datatype then it is called boolen property which may hold true/false value.

E.g boolen z;

**iv. Bound Property :** If the data member of Bean classs will change by changing the value of another Property then this type of variable is called Bound Property. Whenever the value of Bound Property will change then an object of java.beans.PropertyChangeEvent class is generated. To handle this event PropertyChangeListener interface is used.

E.g if salary &job are 2 datamemberof Bean class ,when the value of job properties change,then we will change value of salary Property.

**v. Constraint Property :** If depending upon the value of a data member of the Bean class a condition must be satisfied, then it will be called Constraint Property. If the condition does not satisfy then the Bean class will terminate the execution and the property will not change its value ,such type of property accquires vetopower. Whenever the value of Constraint Property changes then an object of PropertyChangeEvent class is generated and to handle it VetoableChangeListener interface is used.

**b) Method:** The method of Bean class are used to provide and accesss the values of the properties i.e data members.Basically methods are of two types known as :

i- Property Setter/Modifier method
ii- Property Getter/Accesser method

**i) Property accesser method:** This methods are used to fetch the value of Property. This method names must start with a prefix called **get** followed by the property name .The return type of these method must be remain same to that of the data type of the Property , and these method should not accept any parameter.

E.g Let us say the property is String name, then the property setter method is :

public String getName()

**ii. Property modifier method** : This methods are used to modify/set the value of a Property .The name of the method must start with aprefix called **set** followed by the Property name. The return type of this function must always be void. This function must accept a parameter whoose datatype must remain the same to tha of the datatype of the Property.

E.g If name is the pProperty ,then the method will be

public void setName(String b)

**Creating Bean class:** Once we create the Bean class ,then it must be stored in a package & the package must be stored under the classes folder.

**<u>Store it as Mybean.java under classes\pack1 package</u>**

```
package   pack1;
public  class  Mybean {
  String  name, pass;
   public  void  setName(String  n){  name=n;  }
   public  void  setPass(String  p){  pass=p;  }
   public  String getName(){   return name;  }
   public  String  getPass(){  return  pass;  }
};
```

**Using Bean class in jsp:** A jsp file in which we will use the bean class must be use <jsp:use Bean> tag.

**<jsp:useBean>** This tag initialises the Bean class within a jsp file. This tag has the id, class & scope attribues

1. **id:** This attribute will have a user define name as a value so that the user defined name will be considered as the object of Bean class within the jsp file, so that we can invoke the property setter/getter method of the Bean class within the jsp file.
2. **class:** This attribute holds the Bean class name along with its package name i.e., it holds full qualified name of the Bean class.
3. **scope:** This attribute decides the scope of the Bean within a jsp file or for user session or for the entire application. The scope attribute has the following values;
   a. Page
   b. Session
   c. Application

**a) page :** It is the default value of scope attribute, even if we will not provide scope attribute then the default value will be page. Such types of value allows the bean to be used within same page.

**b) session :** This value of the page attribute allows the bean to be used in any web site for an user. Hence, each user will get separate copy of bean object.

**c. application:** This value of the page attribute allows the bean to be used by any web page by any user of the website. Here individual user will get same bean in any web page of the website.

The jsp file in which we will use <jsp:use bean > must use atleast one subtag called <jsp:set property> or< jsp:get property>

**<jsp:set property>:**
This tag is used to provide the value to different properties of the bean class by calling setter().It has the following attributes

a- name
b- property
c- value

a. **name:** This property must hold the name of the object of the bean class.It is the same value given in id attribute of <jsp:use bean>
b. **property:** This attribute must hold the property name of the Bean class to which the value will be provided by setter()
c. **value:** This attribute must hold the value of the property ,and jsp will implicitely call the setter()to provide the value to the property.
Ex. <jsp:useBean id="m1" class="pack1.mybean" scope="page" />
<jsp:set property name="m1"property="name"value="rama"/>in this subtag we are providing the value for the name property.

**<jsp:getProperty>:** This subtag is used to fetch the value of properties of the Bean class.
Ex: <jsp:getProperty name="m1" property ="name" />
In this subtag we are accessing the value of name property ,but here we cant store the value of "name" property, rather the value of "name "property will be printed. If we like to receive /store the value of name property then we may write:
<% String na=m1.getName(); %>
Now rama which is the value of "name"property will be stored in String object na.

**Replacement of<jsp:setProperty>** Generally this subtag is used to provide the value to the properties of the Bean class but if we have several properties then it will be be an overburden to invoke <jsp:setProperty> several times, hence we can use * in the property attribute without using value attribute as <jsp:setProperty name ="m1" property ="*" />
Here * represents all the properties of the Bean class,since we have not givn any value hence the request received by current jsp file must contain the attribute name along with their value in the url or in the request as givn below –(Let us say we have used property="*" i.e above jsp:setPropery in Login.jsp.file)
http://.../login.jsp?name=rama&pass=sama
name and pass in above request must be the textfield name which are equivalent to the properties of the Bean class and this request is available to Login.jsp from Log.html where textfield &password field are given to provide username & password.

## Simple property of a Bean :

Compile Mybean.java in previous page
**Sotre it as BeanDemo.html :**
<html><body bgcolor="yellow"> <h1>
<form action="Bean1.jsp" method ="get">

```
User Name <input type="text" name="t1" ><br>
Password <input type="password" name="p1"><br>
<input  type="submit" value="Submit">
<input  type="reset" value="Reset">
</h1></form></body></html>
```

**Sotre it as Bean1.jsp :**

```
<html><body bgcolor="cyan">
<%!   String s1, s2;   %>
<h1>Setting the property of bean  in jsp<br>
<%
    s1 = request.getParameter("t1");
    s2 = request.getParameter("p1");
%>
<jsp:useBean  id="m1" class="pack1.Mybean" scope="session"  />
<jsp:setProperty name="m1" property="name"  value="<%=  s1 %>"  />
<jsp:setProperty name="m1" property="pass"  value="<%=  s2 %>"  />
<a href="Bean2.jsp">Click </a>
</h1></body></html>
```

**Sotre it as Bean2.jsp :**

```
<html><body bgcolor="magenta">
<h1>Accessing property of bean  in jsp<br>
<jsp:useBean  id="m1" class="pack1.Mybean" scope="session"  />
User Name is <jsp:getProperty name="m1" property="name"    />  <br>
Password is : <jsp:getProperty name="m1" property="pass"    /> <br>
</h1></body></html>
```

## Using  * in the <jsp:setProperty>  :

**Sotre it as BeanDemo.java  under  classes\pack1 :**

```
package   pack1;
public  class  BeanDemo {
    String  name, pass, fname, lname, email;
   int  age;
   public  void  setName(String  p){  name=p;  }
   public  void  setPass(String  p){  pass=p;  }
   public  void  setFname(String  p){ fname=p;  }
   public  void  setLname(String  p){ lname=p;  }
   public  void  setEmail(String  p){ email=p;  }
   public  void  setAge(int  p){ age=p;  }

public  String getName(){   return name;  }
public  String  getPass(){  return  pass;  }
public  String  getFname(){ return fname;  }
public  String  getLname(){ return lname;  }
public  String  getEmail(){ return email;  }
public  int  getAge(){ return age;  }
};
```

**Sotre it as BeanStar.html :**

```
<html><body bgcolor="yellow"> <h1>
<form  action="StarBean1.jsp"  method ="get">
User Name<input type="text" name="name"><br>
Password <input  type="password" name="pass"><br>
First Name <input type="text" name="fname" ><br>
Last name <input type="text" name="lname"><br>
Email <input type="text" name="email" ><br>
Age <input type="text" name="age"><br>
<input  type="submit" value="Submit">
<input  type="reset" value="Reset">
</h1></form></body></html>
```

**Sotre it as StarBean1.jsp :**

```
<html><body bgcolor="cyan">
<h1>Setting the property of bean  in jsp<br>
<jsp:useBean  id="u1" class="pack1.BeanDemo" scope="session"  />
<jsp:setProperty name="u1" property="*"   />
<a href="StarBean2.jsp">Click </a>
</h1></body></html>
```

**Sotre it as StarBean2.jsp :**

```
<html><body bgcolor="magenta">
<h1>Accessing property of bean  in jsp<br>
<jsp:useBean  id="u1" class="pack1.BeanDemo" scope="session"  />
User Name is  <jsp:getProperty name="u1" property="name"    />  <br>
Password is :  <jsp:getProperty name="u1" property="pass"    /> <br>
First Name is : <%=  u1.getFname() %><br>
Last Name is : <%=  u1.getLname() %><br>
Email is : <%=  u1.getEmail() %><br>
Age is :  <%=  u1.getAge() %><br>
</h1></body></html>
```

## Bound property of  a  Bean :

**Store  it  as   BoundBean.java :**

```
package   pack1;
import   java.beans.*;
public  class  BoundBean implements PropertyChangeListener {
   String  job;
   int  sal;
```

```
  PropertyChangeSupport pcs;
  public BoundBean() {
     pcs = new PropertyChangeSupport(this);  //  step -  1
     pcs.addPropertyChangeListener(this);      //  step -  2
 }
  public  void  setJob(String  n){
     String  sold  = job;
      job=n;
      pcs.firePropertyChange("job",sold,  n);   //  step -  3
  }
  public  void  setSal(int  p){  sal=p;  }
  public  String getJob(){   return job;  }
  public  int  getSal(){  return  sal;  }
  public void   propertyChange( PropertyChangeEvent ae) {    //  step -  4
      job= (String) ae.getNewValue();
      if( job.equalsIgnoreCase("manager"))
        sal=30000;
      else  if( job.equalsIgnoreCase("cleark"))
        sal=18000;
      else
        sal=10000;
}};
```

**Store it as BoundBDemo.html :**

```
<html><body bgcolor="yellow"> <h1>
<form  action="BoundBean1.jsp" >
Choose  a  job
<select name="t1" >
   <option>manager</option>
   <option>cleark</option>
   <option>peon</option>
</select>
<br>
<input  type="submit" value="Submit">
<input  type="reset" value="Reset"></h1>
</form></body></html>
```

**Store it as BoundBean1.jsp :**

```
<html><body bgcolor="cyan">
<%!   String s1, s2;   %>
<h1>Using Bound bean in  jsp<br>
<%
        s1 = request.getParameter("t1");
%>
<jsp:useBean  id="u2" class="pack1.BoundBean" scope="session"  />
```

```
<jsp:setProperty name="u2" property="job"  value="<%=  s1 %>"  />
<jsp:setProperty name="u2" property="sal"  />
<a href="BoundBean2.jsp"> get salary</a>
</h1></body></html>
```

**Store it as BoundBean2.jsp :**

```
<html><body bgcolor="magenta">
<h1>Fetching value of bean  in jsp<br>
<jsp:useBean  id="u2" class="pack1.BoundBean" scope="session"  />
The job is :
<jsp:getProperty name="u2" property="job"    />  <br>
Salry is :
<jsp:getProperty name="u2" property="sal"    /> <br>
</h1></body></html>
```

The BoundBean.java has implemented bound property of the bean clas by its ‘job’ & ‘sal’ property, because we want to change the ‘sal’ porperty only when the value of ‘job’ porperty will change.

Whenever the value of ‘job’ porperty will change then an object of PropertyChangeEvent class will be genrated, and to handel this event PropertyChangeListener will be used.

When we click on a Button then an object of ActionEvent class is generated and the ActionListener interface will handel the event. Similarly when the value of ‘job’ porperty will change then an object of PropertyChangeEvent class will be genrated, but we cannot click on ‘job’ property to generate the event, therefore we have to create the event by ourself. It can be done by PropertyChangeSupport class.

The PropertyChangeSupport class must be leanked or related to the bean class, it can be done by passing the object of bean claa in the constructor of PropertyChangeSupport class (as specified by step 1).

As the Button has to be registered so that when the Button will generate the event the listener will handel the event, simillarly the PropertyChangeSupport class needs to be registered by providing the object of the class that has inherited from listener interface (as specified by step 2), here we have given ‘this’ because the current class is inherited from PropertyChangeListener interface.

When the Button is clicked then an object of ActionEvent class is genrated but here we have to generate the event by firePropertyChange() of the PropertyChangeSupport class(as specified by step 3) by providing the property name as the 1st paramter, the 2nd & 3rd parameter must provide the old and new value of the property. When this method will execute then an object of PropertyChangeEvent will be generated. To handel this event the propertyChange() of PropertyChangeListener will be used. Hence the value of sal property must be changed within this method

## Contraint property of a Bean :

**Store it as ConstBean.java :**

```
package   pack1;
import   java.beans.*;
```

```
public  class ConstBean  implements VetoableChangeListener {
   String  pass;
   VetoableChangeSupport  vcs;
   public ConstBean() {
      vcs = new VetoableChangeSupport(this);
      vcs.addVetoableChangeListener(this);
  }
   public  void  setPass(String  n){
      String  sold  = pass;
       pass = n;
       try {
       vcs.fireVetoableChange("pass",sold,  n);
       }catch(Exception e){
            System.out.println(e);
        }
   }
   public  String getPass(){   return  pass;  }
   public void   vetoableChange( PropertyChangeEvent ae) throws PropertyVetoException {
        String nm= (String) ae.getNewValue();
        if(nm.length() < 8) {
                    PropertyVetoException   pv= new  PropertyVetoException("password  cannot  be  less  than  8
charcter", ae);
                    pass ="***  password is less than 8 character";
                    throw pv;
        }
}};
```

**<u>Store it as ConstDemo.html :</u>**

```
<html><body bgcolor="yellow"> <form  action="Const1.jsp" ><h1>
Enter ur name   <input type="text" name="t1" ><br>
Enter ur password   <input type="password" name="t2" ><br>
<input  type="submit" value="Submit">    <input  type="reset" value="Reset">
</h1> </form></body></html>
```

**<u>Store it as Const1.jsp :</u>**

```
<html><body bgcolor="magenta">
<h1>Using const bean  in jsp<br>
<% String s1, s2;  %>
<%
   s1= request.getParameter("t1");
   s2= request.getParameter("t2");
%>
<jsp:useBean  id="u3" class="pack1.ConstBean" scope="session"  />
Password is :
```

```
<jsp:setProperty name="u3" property="pass"  value="<%= s2 %>" />  <br>
<jsp:getProperty name="u3" property="pass"    />
<br></h1></body></html>
```

When the property of a bean will change depending upon a condition then it is called constraint bean. If the condition will satisfy then the bean will change the value of the property otherwise the bean will terminate the execution, such type of bean acquires veto power.

## User defined Tag/Custom Tag

This is a process of creating user defined tag. This tag used to perform any kind of task. It provides the following advantages-

1) **Code reusability**:- After the creation of tag it can be used by any no. of JSP to perform the same tag. Hence the custom tag class or the code behind the tag becomes reusable.
2) **Code abstraction**:- User of the tag shouldn't know the implementation of logic of the custom tag which allow a programmer to remove the task from the JSP.
3) **Avoiding duplication**:- A custom tag may perform a common task required for more than jsp. Hence the required code to perform the common task can be kept within the tag class.

**Requirement for creating Custom tag:-**

a) **Tag class :-** This class contains logic for the tag.
b) **TLD (Tag library discrepter):-** As the info^ of the servlet is given in "web.xml" file, similarly the info of custom tag is given in .TLD file. It's a xml type file.

As in jsp we use <jsp:useBean> tag to get information about the Bean class, similarly in the jsp file use taglib attribute of page directive ( <%@ taglib...>) . to acess the tag class as well as o create the tag name (tag name is the similar to object of Bean class in jsp)

**c) Creation process of tag class:-**

(i) Create a class by inheriting from TagSupport of class available in javax.servlet.jsp.tagext package and the user defined class must remain as public.
(ii) Provide variables within this class which will become attributes of the custom tag.
(iii) The setter ( ) is used to provide the value of attributes(variables).
(iv) Override the method of TagSupport of class to represent the logic for <start> and <end>

For starting tag the method is public int doStartTag ( ), for ending tag the method is public int doEndTag(). This method return an integer value by the constant present in TagSupport of the class.
These constants are:-

1) TagSupport**.**EVAL_ BODY_ INCLUDE : This constant can be applied only as the return value of doStartTag ( ). This makes the body of the tag to execute (the content available within starting & ending tag in jsp file will execute)
2) TagSupport**.**SKIP_BODY : This constant can be applied only to doStartTag ( ). It doesn't allow the body of tag to execute.
3) TagSupport**.**EVAL_PAGE : This constant can only be used for doEndTag ( ). This makes the remaining part of the page to execute.

4) TagSupport.SKPI_PAGE : This constant can be applied only for doEndTag ( ). This doesn't allow remaining part of the page to execute.

d) Create an object of JspWriter class by using getOut ( ) of pageContext object. This is required for printing the o/p lines of the tag from doStartTag ( ) or doEndTag ( ).
e) The user defined tag class must be stored within a package , and the package must be stored under classes folder of WEB-INF folder of root directory.
f) Compile the user defined Tag class by providing servlet.jar in class path.

**Creation process of TLD:-**
The extension of this file must be .TLD within any name. Copy any .TLD file available in Tomcat home directory and paste it under WEB-INF folder. Open this file in edit mode and remove all the tag within </short_name> and </taglib>. Use the following tags before </taglib> but after </short name> as given below :-
…………………… </short_name>
<tag> - it is used to provide info of user defined tag class.
<name> - it is name by which jsp will access the user defined tag class. (it is similar to the url pattern of servlet.) </name>
<tag-class> Provie the full qulaified name of the custom tag class </tag-class>
<attribute> - it is used to provide info about the attribute of tag.
<name> col </name> - name of attribute of tag, and it must be same name as that of the data member of user defined tag class.
<required> true </required> - if value of this is true then this attribute of tag is a mandatory attribute, if false then this attribute is optional.
<rtexprvalue> true <rtexprvalue> - if it is true then the attribute will accept value dynamically at run-time, such as by expression tag. If it is false then the attribute will not accept run-time value.
</attribute>
<attribute> - info of 2nd attribute
<name> ……… </name>
<required> ……………..</reqired>
<rtexprvalue>…………</rtexprvalue>
</attribute>
</tag>

**<u>Using the user defined Tag / custom tag in Jsp:-</u>**
As Jsp uses <jsp:useBean> to use a bean class similarly to use the <custom> the Jsp must provide taglib directive tag as given below
<%@ taglib uri =”/WEB_INF/Test.tld” prefix = “p1” %>
The **uri** attribute of taglib directive gets the info of custom tag from .tld file. Hence it must hold the name of the .tld file along with the WEB-XML folder. Because Jsp file is stored in context folder.
**prefix** attribute holds an user defined name, which will behave like an object of .tld file by which we can access different tag names available in .tld file.

**Store it as hcol.java under MyTags package of classes folder:**

```
package  MyTags;
import  java.io.*;
import   javax.servlet.*;
import   javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class hcol  extends  TagSupport {
  public  String col;
  public  String  sz;
  public  int  doStartTag() throws JspException {
    try {
      JspWriter out = pageContext.getOut();
   out.println("<font color=" + col +" size="+sz+">");
  }catch(IOException e) {    }
   return  EVAL_BODY_INCLUDE;
  }
  public  int doEndTag() throws JspException {
  try {
   JspWriter out= pageContext.getOut();
   out.write("</font>");
  }catch(IOException e){  }
  return EVAL_PAGE;
  }
  public  void  setcol(String c) {
      col=c;
}
  public  void  setsz(String c) {
      sz=c;
}};
```

**Test.tld :**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
<short-name>hello</short-name>

<tag>
   <name>h7</name>   <tag-class>MyTags.hcol</tag-class>
<attribute>
  <name>col</name>
  <required>true</required>  <rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
  <name>sz</name>
  <required>true</required>  <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
</taglib>
```

**Tagattrib.jsp :**

```
<%@ taglib  uri="/WEB-INF/Test.tld"    prefix="p1" %>
<html><body bgcolor="pink">Using the custom tag<br><form>
Enter : <br>Name <input type="text" name="t1"><br>
Color <input type="text" name="t2"><br>
Size <input type="text" name="t3"><br>
<input  type="submit" value="Ok">
 </form>
<%
String  s1= request.getParameter("t1");
String   s2=  request.getParameter("t2");
String   s3=  request.getParameter("t3");
if(s1 != null) {
%>
<p1:h7 col="<%= s2 %>"  sz="<%= s3 %>">
&nbsp; <%= s1 %>
</p1:h7>
<%  }  %>
</body></html>
```

**PageContext:** PageContext represents a class that provides methods that are implementation-dependent. The PageContext class itself is abstract, so in the _ jspService method of a JSP servlet class, a PageContext object is obtained by calling the getPageContext method of the JspFactory class. The PageContext class provides methods that are used to create other objects. For example, its getOut method returns a JspWriter object that is used to send strings to the web browser. Other methods that return servlet-related objects include the following:

• getRequest, returns a ServletRequest object
• getResponse, returns a ServletResponse object
• getServletConfig, returns a ServletConfig object
• getServletContext, returns a ServletContext object
• getSession, returns an HttpSession object

**The JspWriter Class :** The JspWriter class is derived from the java.io.Writer class and represents a Writer that you can use to write to the client browser. Of its many methods, the most important are the print and println methods. Both provide enough overloads that ensure you can write any type of data.

Additional methods allow you to manipulate the buffer. For instance, the clear method clears the buffer. It throws an exception if some of the buffer's content has already been flushed. Similar to clear is the clearBuffer method, which clears the buffer but never throws any exception if any of the buffer's contents have been flushed.

## Accessing a Jsp from the Servlet.

**ServRec.html :**

```
<html> < body bgcolor = “cyan”> <h1>
<form action = “../servJsp” method = get>
Name : <input type = “text” name = “t1”> <br>
Password : <input type = “password” name = “t2”><br>
<input type = “submit” value =”submit”>
</hi> </form> </body> </html>
```

The above html file is accessing the Serv2jsp.java (servlet) whose url pattern is servJsp in web.xml. but this ServRec.html file is not stored in context folder (i.e./under google) rather the ServRec.html is stored under c:\....\webapp\google\serv folder, hence we must provide the root folder name in order to access the srevlet. Since the serv folder is within google folder hence in action we have given **../ServJsp**, here **..** refers to parent folder i.e. to the google.

**Serv2Jsp.java**

```
Import  java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Serv2Jsp extends HttpServlet {
  public void doGet (HttpServletRequest request, HttpServletResponse response)  throws IOException,
ServletException {
    try {
        request.setAttribute ("s1",  "Serv2Jsp");            //Set the attribute & value to forward to hello.jsp
        ServletConfig scf = getServletConfig();
        ServletContext sc =scf.getServletContext();
        RequestDispatcher rd = sc.getRequestDispatcher("/serv/hello.jsp");
         rd.forward(request, response);
    } catch (Exception ex) {
         ex.printStackTrace ();
    }
}};
```

**Hello.jsp :**

```
<html> <body bgcolor=”pink”> <h1>
1 have been invoked by
<%  out.print(request.getAttribute(“s1”).toString() );   %>
Servlet .<br> your name : <%=  request.getParameter(“t1”)  %><br>
your password : <%=  request.getParameter(“t2”)   %>
<h1> </html>
```

**getServletConfig( ) :-** This method GenericServlet class returns a reference of ServletConfig interface. This interface is used by a servlet container to pass info to a servlet during initialization. Since this interface we have used in above servlet, hence the value will be the passed to the Hello.jsp by the web server. Because this servlet program is forwarding the users request to the Hello.Jsp. therefore the value given by sevlet container by using ServletConfig interface, apart from the users request exra information can be passed from srvlet to Hello.Jsp by setAttribute ().

**setAttribute (String variable name,String variable value) :** This method of HttpServletRequest interface is used to provide extra value from the servlet to other servlet/Jsp. This method will accept the user defined variable name along with a value as pecified by 1st & 2nd parameter respectively, so that the other servlet/jsp will access the value by providing the variable name in getAttribute() of HttpServletRequest interface in servlet/by request object in Jsp file.

## Using a plugin in a jsp file :

The **<jsp:plugin>** tag is used to executes an applet or Bean in jsp and if necessary this tag will download a Java plug-in(java enviornment like appletviewer or jvm) to execute the applet or bean.

**Store it as Clock.java :**

```
import java.util.*;
import java.awt.*;
import java.applet.*;
import java.text.*;
/*<applet code="Clock" width="300" height="200"></applet> */
public class Clock extends Applet implements Runnable {
  SimpleDateFormat sd;
  String  dt, tm;
    public void init() {
        setBackground(Color.magenta);
        setForeground(Color.blue);
        setFont(new Font("aSFDa", Font.BOLD,40));
        Thread tt = new Thread(this);
        tt.start();
    }
    public void run() {
        while (true) {
            try {
                Thread.currentThread().sleep(1000);
            } catch (InterruptedException e) {   }
            sd = new SimpleDateFormat ("hh:mm:ss");
            tm = sd.format(new Date());
            sd = new SimpleDateFormat ("dd-MMM-yyyy");
            dt = sd.format(new Date());
            repaint();
        }
    }
 public void  paint(Graphics g) {
     g.drawString(tm, 50,50);
     g.drawString(dt, 50,120);
}};
```

**Store it as Plugin.jsp :**

```
<html>
<title> Plugin example </title>
<body bgcolor="cyan">
<h3> Current time is : </h3>
<jsp:plugin type="applet" code="Clock.class"  codebase="applet"   jreversion="1.2"  width="200"
height="150" >
   <jsp:fallback>
      Plugin tag OBJECT or EMBED not supported by browser.
   </jsp:fallback>
</jsp:plugin>
<p><h4><font color=red>The above applet is loaded using the Java Plugin from a jsp page using the plugin
tag.</font></h4></body></html>
```

**Attributes of <jsp:plugin>**

**type = "bean/applet"** -- The type of object the plug-in will execute. You must specify either bean/applet, as this attribute has no default value.

**code = "class file name"** -- The name of java class file the plug-in will execute. You must include the .class extension in the name. The file you specify should be in the directory named in the codebase attribute.

**codebase = "class file directory name"** -- The directory (or path to the directory) that contains the java class file the plug-in will execute. If you don't supply a value, the path of Jsp file that calls <Jsp.plugin> is used.

**name = "instance Name"** -- A name for the instance of the Bean or applet. Which makes it possible for applet or Beans called by the same Jsp file to communicate with each other.

**<jsp:params>**

```
        <jsp:params name =”parameter Name”  value = “{parameter value/<% =expression%>}   ”/>
</jsp:params>
```

The parameters and value that you want to pass to the applet or bean. To specify more than one <jsp:param> we have to specify one <jsp:param> element within the other <jsp:params> elements. The name attribute specifies the parameter name and takes a case-sensitive literal string. The value attribute specifes the parameter value and takes either a case-sensitive literal string or an expression that is evaluated at runtime.

**<jsp:fallback>** text msg for user **</jsp:fallback>** -- A text msg to display for the user if the plug-in can't be started. If the plug-in starts but the applet or bean doesn't, the plug-in usually displays a popup window explaining the error.