JSP(JAVA SERVER PAGES)

The main purpose of the web applications in Enterprise applications area is to generate dynamic response from server machine.

To design web applications at server side we may use Web Technologies like CGI, Servlets, JSP and so on.

CGI is basically a Process based technology because it was designed on the basis of C technology.

If we deploy any CGI application at server then for every request CGI container will generate a separate process.

In the above context, if we send multiple number of requests to the same CGI application then CGI container has to generate multiple number of processes at server machine.

To handle multiple number of processes at a time server machine has to consume more number of system resources, as a result the performance of the server side application will be reduced.

To overcome the above problem we have to use Thread based technology at server side like servlets.

In web application development, servlets are very good at the time of pick up the request and process the request but servlets are not good at the time of generating dynamic response to client.

Servlet is a Thread based technology, if we deploy it at server then container will create a separate thread instead of the process for every request from the client.

Due to this Thread based technology at server side server side application performance will be increased.

In case of the servlet, we are unable to separate both presentation logic and business logic.

If we perform any modifications on servlets then we must perform recompilation and reloading.

If we want to design web applications by using servlets then we must require very good knowledge on Java technology.

JSP is a server side technology provided by Sun Microsystems to design web applications in order to generate dynamic response.

The main intention to introduce Jsp technology is to reduce java code as much as possible in web applications.

Jsp technology is a server side technology, it was designed on the basis of Servlet API and Java API.

In web application development, we will utilize Jsp technology to prepare view part or presentation part.

Jsp technology is very good at the time of generating dynamic response to client with very good look and feel.

If we want to design any web application with Jsp technology then it is not required to have java knowledge.

In case of Jsp technology, we are able to separate presentation logic and business logic because to prepare presentation logic we will use html tags and to prepare business logic we will use Jsp tags separately.

If we perform any modifications on Jsp pages then it is not required to perform recompilation and reloading because Jsp pages are auto-compiled and auto-loaded.

Jsp Deployment:

In web application development, it is possible to deploy the Jsp pages at any location of the web application directory structure, but it is suggestible to deploy the Jsp pages under application folder.

If we deploy the Jsp pages under application folder i.e. public area thenwe are able to access that Jsp page from client by using its name directly in the url.

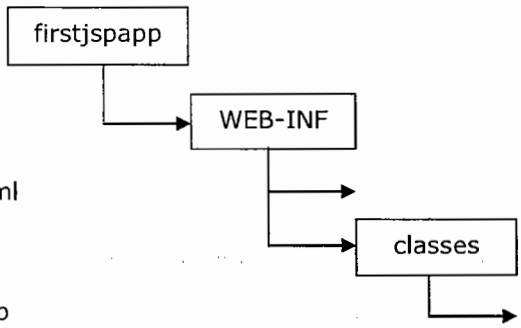
If we deploy the Jsp pages under private area(WEB-INF, classes) then we must define url pattern for the Jsp page in web.xml file and we are able to access that Jsp page by specifying url pattern in client url.

To configure Jsp pages in web.xml file we have to use the following xml tags.

```
<web-app>
        ----------
<servlet>
<servlet-name>logical_name</servlet-name>
<Jsp-file>context relative path of Jsp page</Jsp-file >
</servlet>
<servlet-mapping>
<servlet-name>logical_name</servlet-name>
<url-pattern>pattern_name</url-pattern>
</servlet-mapping>
          ----------
</web-app>
```

Application:

Directory Structure:



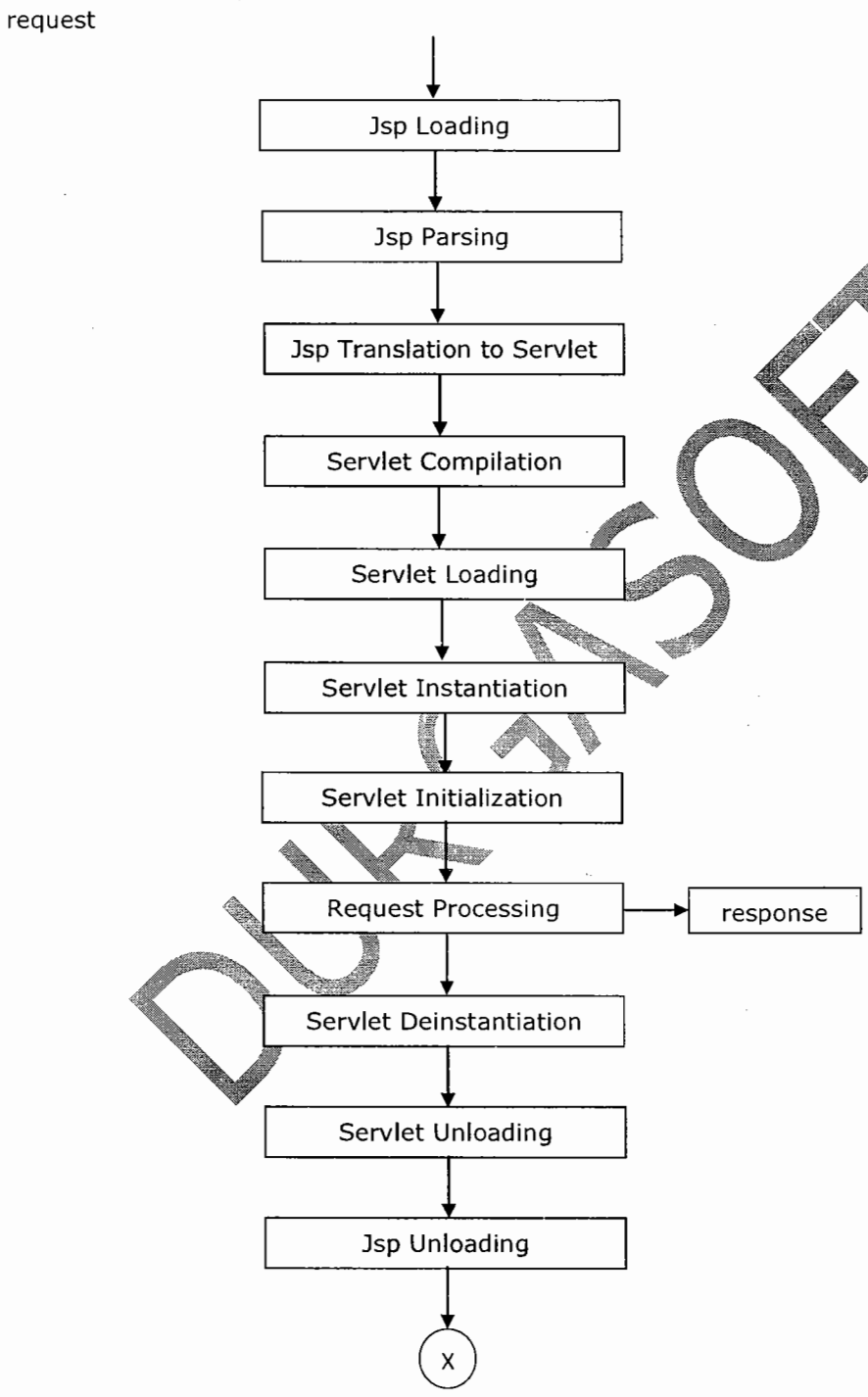
first.jsp:

```
<html>
<body bgcolor="lightgreen">
<center><b><font size="7" color="red">
<br><br>
   First Jsp Application Deployed in Classes
</font></b></center>
</body>
</html>
```

Web.xml:

```
<web-app>
<servlet>
<servlet-name>fj</servlet-name>
<jsp-file>/WEB-INF/classes/first.jsp</jsp-file>
</servlet>
<servlet-mapping>
<servlet-name>fj</servlet-name>
<url-pattern>/jsp</url-pattern>
</servlet-mapping>
</web-app>
```

Jsp Life Cycle:



4

When we send request from client to server for a particular Jsp page then container will pick up the request, identify the requested Jsp pages and perform the following life cycle actions.

1. Jsp Loading:

Here container will load Jsp file to the memory from web application directory structure.

2. Jsp Parsing:

Here container will check whether all the tags available in Jsp page are in well-formed format or not.

3. Jsp Translation to Servlet:

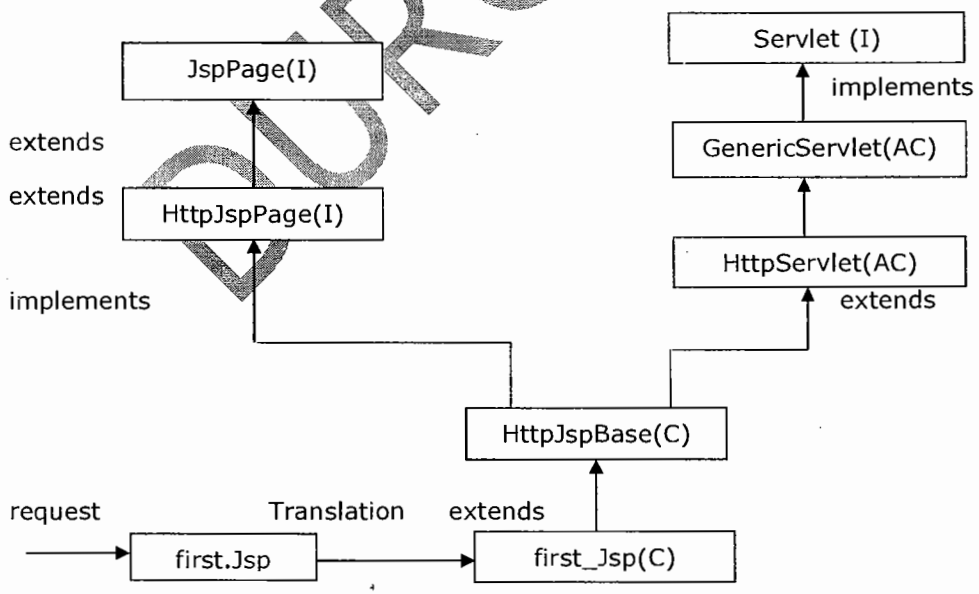
After the Jsp parsing container will translate the loaded Jsp page into a particular servlet.

While executing a Jsp page Tomcat container will provide the translated servlet in the following location at Tomcat Server.

C;\Tomcat7.0\work\catalina\localhost\org\apache\Jsp\first_Jsp.java

If the Jsp file name is first.jsp then Tomcat Server will provide a servlet with name first_jsp. By default all the translated servlets provided by Tomcat container are final.

The default super class for translated servlet is **HttpJspBase**.



Where **JspPage** interface has declared the following methods.

public void _JspInit()

public void _JspDestroy()

Where **HttpJspPage** interface has provided the following method.

public void _JspService(HttpServletRequest req, HttpServletResponse res)

For the above 3 abstract methods HttpJspBase class has provided the default implementation but _JspService(_,_) method would be overridden in first_jsp class with the content what we provided in first.jsp file.

4. Servlet Compilation:

After getting the translated servlet container will compile servlet java file and generates the respective .class file.

5. Servlet Loading:

Here container will load the translated servlet class byte code to the memory.

6. Servlet Instantiation:

Here container will create object for the loaded servlet.

7. Servlet Initialization:

Here container will access _JspInit() method to initialize the servlet.

8. Creating request and response objects:

After the servlet initialization container will create a thread to access _JspService(_,_) method, for this container has to create HttpServletRequest and HttpServletResponse.

9. Generating Dynamic response:

After getting request and response objects container will access _JspService(_,_) method, by executing its content container will generate some response on response object.

10. Dispatching Dynamic response to Client:

When container generated thread reached to the ending point of _JspService(_,_) method then that thread will be in Dead state, with this container will dispatch dynamic response to client through the Response Format prepared by the protocol.

11. Destroying request and response objects:

When the dynamic response reached to client protocol will terminate its virtual socket connection, with this container will destroy request and response objects.

12. Servlet Deinstantiation:

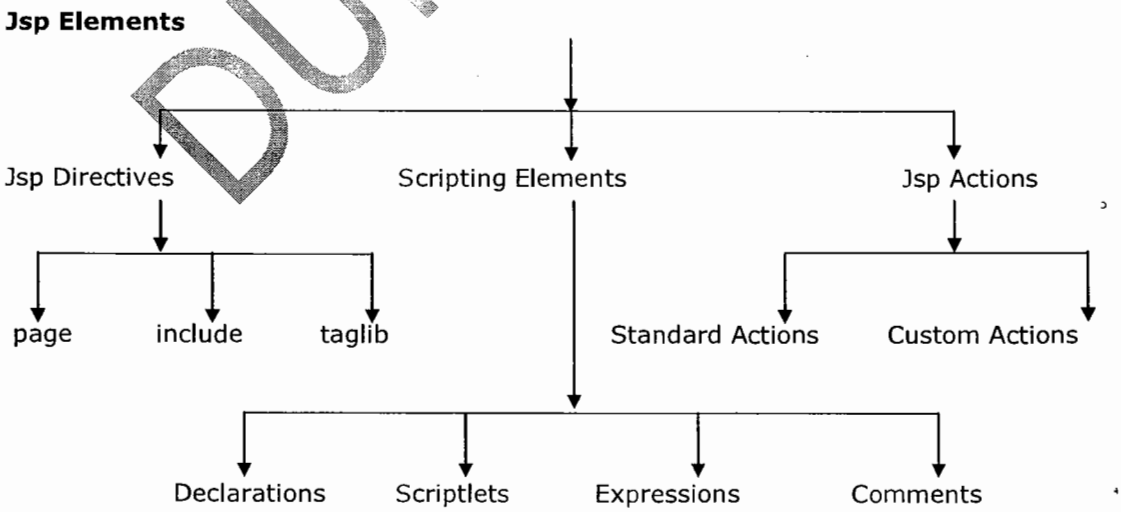
After destroying request and response objects container will be in waiting state depends on the container, then container identifies no further request for the same resource then container will destroy servlet object, for this container will execute _JspDestroy() method.

13. Servlet Unloading and Jsp Unloading:

After the servlet deinstantiation container will eliminate the translated servlet byte code and Jsp code from memory.

Jsp Elements:

In web applications to design Jsp pages we have to use the following elements.



Q: What are the differences between Jsp Directives and Scripting Elements?

Ans: 1. In web applications, Jsp Directives can be used to define present Jsp page characteristics, to include the target resource content into the present Jsp page and to make availableuser defined tag library into the present Jsp page.

In web applications, Jsp Scripting Elements can be used to provide code in Jsp pages.

2. All the Jsp Directives will be resolved at the time of translating Jsp page to servlet.

All the Jsp Scripting Elements will be resolved at the time of request processing.

3. Majority of the Jsp Directives will not give direct effect to response generation, but majority of Scripting Elements will give direct effect to response generation.

Q: To design Jsp pages we have already Jsp Scripting Elements then what is requirement to go for Jsp Actions?

Ans: In Jsp applications, Scripting Elements can be used to allow java code inside Jsp pages but the main theme of Jsp technology is not to allow java code inside the Jsp pages.

In the above context, to preserve the theme ofJsp technology we have to eliminate scripting elements from Jsp pages, for this we have to provide an alternative i.e. Jsp Actions provided by Jsp technology.

In case of Jsp Actions, we will define scripting tag in place of java code, in Jsp pages and we will provide the respective java code inside the classes folder.

In this context, when Jsp container encounter the scripting tag then container will execute the respective java code and perform a particular action called as **Jsp Action.**

1. Jsp Directives:

To provide Jsp Directives in Jsp pages we have to use the following syntaxes.

1. Jsp-Based Syntax:

<%@Directive_name [attribute-list]%>

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

2. XML-Based Syntax:

<jsp:directive.directiveName[attribute-list]%/>

Ex: <jsp:directive.page import="java.io.*"/>

There are 3 types of Directives in Jsp technology.

1. Page Directive
2. Include Directive
3. Taglib Directive

1. Page Directive:

In Jsp technology, **Page Directive** can be used to define the present Jsp page characteristics like to define import statements, specify particular super class to the translated servlet, to specify metadata about present Jsp pages and so on.

Syntax 1:<%@page [attribute-list]%>

Syntax 2:<jsp:directive.page [attribute-list]/>

Where attribute-list in Jsp page directive may include the following list.

1. language
2. contentType
3. import
4. extends
5. info
6. buffer
7. autoFlush
8. errorPage
9. isErrorPage
10. session
11. isThreadSafe
12. isELIgnored

1. language:

This attribute can be used to specify a particular scripting language to use scripting elements.

The default value of this attribute is java.

Ex: <%@page language="java"%>

2. contentType:

This attribute will take a particular MIME type in order to give an intimation to the client about to specify the type of response which Jsp page has generated.

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

3. import:

This attribute can be used to import a particular package/packages into the present Jsp pages.

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

If we want to import multiple number of packages into the present Jsp pages then we have to use either of the following 2 approaches.

Approach 1:

Specify multiple number of packages with comma(,) as separator to a single import attribute.

Ex: <%@page import="java.io.*,java.util.*,java.sql.*"%>

Approach 2:

Provide multiple number of import attributes for the list of packages.

Ex: <%@page import="java.io.*" import="java.util.*" import="java.sql.*"%>

Note: Among all the Jsp page attributes only import attribute is repeatable attribute, no other attribute is repeatable.

The default values of this attribute are java.lang, javax.servlet, javax.servlet.http, javax.servlet.jsp.

4. extends:

This attribute will take a particular class name, it will be available to the translated servlet as super class.

Ex: <%@page extends="com.dss.MyClass"%>

Where MyClass should be an implementation class to HttpJspPage interface and should be a subclass to HttpServlet.

The default value of this attribute is HttpJspBase class.

5. info:

This attribute can be used to specify some metadata about the present Jsp page.

Ex: <%@page info="First Jsp Application"%>

If we want to get the specified metadata programmatically then we have to use the following method from Servlet interface.

public String getServletInfo()

The default value of this attribute is Jasper JSP2.2 Engine.

6. buffer:

This attribute can be used to specify the particular size to the buffer available in

JspWriter object.

Note: Jsp technology is having its own writer object to track the generated dynamic response, JspWriter will provide very good performance when compared with PrintWriter in servlets.

Ex: <%@page buffer="52kb"%>

The default value of this attribute is 8kb.

7. autoFlush:

It is a boolean attribute, it can be used to give an intimation to the container about to flush or not to flush dynamic response to client automatically when JspWriter buffer filled with the response completely.

If autoFlush attribute value is true then container will flush the complete response to the client from the buffer when it reaches its maximum capacity.

If autoFlush attribute value is false then container will raise an exception when the buffer is filled with the response.

Ex:
```
<%@page buffer="52kb" autoFlush="true"%>
<html>
<body bgcolor="lightgreen">
<center><b><font size="7" color="red"><br><br>
<%
    for(int i=0; i<10000000; i++) {
        out.println("RAMA");
    }
    %>
</font></b></center></body>
</html>
```

In the above piece of code, if we provide autoFlush attribute value false then container will raise an exception like
org.apache.jasper.JasperException:An exception occured processing JSP page/first.jsp at line:9

root cause: java.io.IOException:Error:Jsp Buffer Overflow.

Note: if we provide 0kb as value for buffer attribute and false as value for autoFlush attribute then container will raise an exception likeorg.apache.jasper.JasperException:/first.jsp(1,2) jsp.error.page.badCombo

The default value of this attribute is true.

8. errorPage:

This attribute can be used to specify an error page to execute when we have an exception in the present Jsp page.

Ex: <%@page errorPage="error.jsp"%>

9. isErrorPage:

It is a boolean attribute, it can be used to give an intimation to the container about to allow or not to allow exception implicit object into the present Jsp page.

If we provide value as true to this attribute then container will allow exception implicit object into the present Jsp page.

If we provide value as false to this attribute then container will not allow exception implicit object into the present Jsp page.

The default value of this attribute is false.

Ex: <%@page isErrorPage="true"%>

first.jsp:

```
<%@page errorPage="error.jsp"%>
<%
java.util.Date d=null;
out.println(d.toString());
%>
```

error.jsp:

```
<%@page isErrorPage="true"%>
<html>
<body bgcolor="lightgreen">
```

```
<center><b><font size="" color=""><br><br>
<%=expression%>
</font></b></center></body>
</html>
```

10. session:

It is a boolean attribute, it is give an intimation to the container about to allow or not to allow session implicit object into the present Jsp page.The default value of this attribute is true.

Ex: <%@page session="true"%>

11. isThreadSafe:

It is a boolean attribute, it can be used to give an intimation to the container about to allow or not to allow multiple number of requests at a time into the present Jsp page.

If we provide true as value to this attribute then container will allow multiple number of requests at a time.

If we provide false as value to this attribute then automatically container will allow only one request at a time and it will implement SingleThreadModel interface in the translated servlet.

The default value of this attribute is true.

Ex: <%@page isThreadSafe="true"%>

12. isELIgnored:

It is a boolean attribute, it can be used to give an intimation to the container about to allow or not to allow Expression Language syntaxes in the present Jsp page.

Note: Expression Language is a Scripting language, it can be used to eliminate java code completely from the Jsp pages.

If isELIgnored attribute value is true then container will eliminate Expression Language syntaxes from the present Jsp page.

If we provide false as value to this attribute then container will allow Expression Language syntaxes into the present Jsp pages.

The default value of this attribute is false.

Ex: <%@page isELIgnored="true"%>

2. Include Directive:

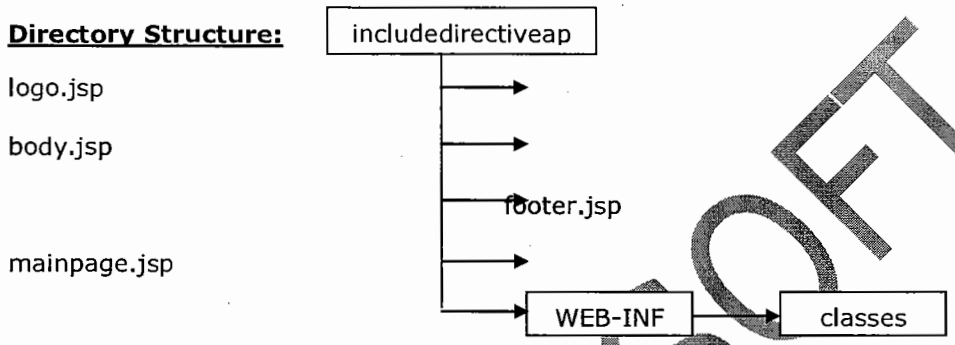
Include Directive can be used to include the content of the target resource into the present Jsp page.

Syntax:<%@include file="--"%>

Where file attribute can be used to specify the name and location of the target resource.

Application:

Directory Structure:



logo.jsp:

```
<html>
<body><center>
<table width="100%" height="20%" bgcolor="red">
<tr><td colspan="2"><center><b><font size="7" color="white">
Durga Software Solutions
</font></b></center></td></tr>
</table></center></body>
</html>
```

footer.jsp:

```
<html>
<body><center>
<table width="100%" height="15%" bgcolor="blue">
<tr><td colspan="2"><center><b><font size="6" color="white">
copyrights2010-2020@durgasoftwaresolutions
</font></b></center></td></tr>
</table></center></body>
</html>
```

body.jsp:

```
<html>
<body bgcolor="lightyellow">
<center><b><font size="7">
<p><br>
   Durga Software Solutions is one of the Training Institute.
<br><br></p>
</font></b></center></body>
</html>
```

mainpage.jsp:

```
<%@include file="logo.jsp"%>
<%@include file="body.jsp"%>
<%@include file="footer.jsp"%>
```

3. Taglib Directive:

The main purpose of **Taglib Directive** is to make available user defined tag library into the present Jsp pages.

Syntax:<%@taglib uri="_ _" prefix="_ _"%>

Where uri attribute can be used to specify the name and location of user defined tag library.

Where prefix attribute can be used to define prefix names for the custom tags.

Ex: <%@taglib uri="/WEB-INF/db.tld" prefix="connect"%>

2. Scripting Elements:

The main purpose of **Jsp Scripting Elements** is to allow java code directly into the present Jsp pages.

There are 3 types of Scripting Elements.

1. Declarations
2. Scriptlets
3. Expressions

1. Declarations:

It can be used to provide all the java declarations like variable declarations, method definitions, classes declaration and so on.

Syntax:<%!

Java Declarations
%>

If we provide any java declarations by using declaration scripting element then that all java declarations will be available to the translated servlet as class level declarations.

2. Scriptlets:

This Scripting Element can be used to provide a block of java code.

Syntax:<%

Block of Java code
%>

If we provide any block of java code by using scriptlets then that code will be available to translated servlet inside _jspService(_,_) method.

3. Expressions:

This is scripting element can be used to evaluate the single java expression and display that expression resultant value onto the client browser.

Syntax:<%=Java Expression%>

If we provide any java expression in expression scripting element then that expression will be available to translated servlet inside _JspService(_,_) method as a parameter to out.write() method.

Ex: out.write(Java Expression);

first.jsp:

```
<%@page import="java.util.*"%>
<%!
Date d=null;
String date=null;
%>
<%
d=new Date();
date=d.toString();
%>
<html>
<body bgcolor="lightyellow">
<center><b><font size="6" color="red"><br><br>
Today Date : <%=date%>
</font></b></center></body>
</html>
```

Translated Servlet:

```
-------------
-------------
import java.uitl.*;
public final class first_jsp extends HttpJspBase implements JspSourceDependent
{
   Date d=null;
   String date=null;
public void _jspInit()throws ServletException
   {  }
public void _jspDestroy()
   {  }
public void _jspService(HttpServletRequest req, HttpServletResponse res)throws SE, IOE
   {
      d=new Date();
      date=d.toString();
      out.write("<html>");
      out.write("<body bgcolor='lightyellow'>");
      out.write("<center><b><font size='6' color='red'><br><br>");
      out.write("Today Date.....");
      out.write(date);
      out.write("</font></b></center></body></html>");
}
}
```

4. Jsp Comments:

In Jsp pages, we are able to use the following 3 types of comments.

1. XML-Based Comments
2. Jsp-Based Comments
3. Java-Based Comments inside Scripting Elements

1. XML-Based Comments:

```
<!--
      --------
      --------   Description
      --------
   -->
```

2. Jsp-Based Comments:

```
<%--
        --------
        --------   } Description
        --------
    --%>
```

3. Java-Based Comments inside Scripting Elements:

1. Single Line Comment:

```
//--------Description---------
```

2. Multiline Comment:

```
/*
        --------
        --------   } Description
        --------
    */
```

3. Documentation Comment:

```
/**
        --------
        --------   } Description
        --------
*/
```

Jsp Implicit Objects:

In J2SE applications, for every java developer it is common requirement to display data on command prompt.

To perform this operation every time we have to prepare PrintStream object with command prompt location as target location

```
PrintStream ps=new PrintStream("C:\program Files\.....\cmd.exe") ;

ps.println("Hello");
```

In java applications, PrintStream object is frequent requirement so that java technology has provided that PrintStream object as predefined object in the form of out variable in System class.

```
public static final PrintStream out;
```

Similarly in web applications, all the web developers may require some objects like request, response, config, context, session and so on are frequent requirement, to get these objects we have to write some piece of java code.

Once the above specified objects are as frequent requirement in web applications, Jsp technology has provided them as predefined support in the form of Jsp Implicit Objects in order to reduce burden on the developers.'

Jsp technology has provided the following list of implicit objects with their respective types.

1. out ------→ javax.servlet.jsp.JspWriter
2. request ------→ javax.servlet.HttpServletRequest
3. response ------→ javax.servlet.HttpServletResponse
4. config ------→ javax.servlet.ServletConfig
5. application ------→ javax.servlet.ServletContext
6. session ------→ javax.servlet.http.HttpSession
7. exception ------→ java.lang.Throwable
8. page ------→ java.lang.Object
9. pageContext ------→ javax.servlet.jsp.PageContext

Q: What is the difference between PrintWriter and JspWriter?
Ans: PrintWriter is a writer in servlet applications, it can be used to carry the response. PrintWriter is not BufferedWriter so that its performance is very less in servlets applications.

JspWriter is a writer in Jsp technology to carry the response. JspWriter is a BufferedWriter so that its performance will be more when compared with PrintWriter.

Q: What is PageContext? and What is the purpose of PageContext in Jsp Applications?

Ans: PageContext is an implicit object in Jsp technology, it can be used to get all the Jsp implicit objects even in non-jsp environment.

To get all the Jsp implicit objects from PageContext we have to use the following method.

public Xxx getXxx()

Where Xxx may be out, request, response and so on.

Ex: JspWriterout=pageContext.getOut();

HttpServletRequest request=pageContext.getRequest();

ServletConfig config=pageConetxt.getServletConfig();

Note: While preparing Tag Handler classes in custom tags container will provide pageContext object as part of its life cycle, from this we are able to get all the implicit objects.

In Jsp applications, by using pageContext object we are able to perform some operations with the attributes in Jsp scopes page, request, session and application like adding an attribute, removing an attribute, getting an attribute and finding an attribute.

To set an attribute onto a particular scope pageContext has provided the following method.

public void setAttribute(String name, Object value, int scope)

Where scopes may be

public static final int PAGE_SCOPE=1;
public static final int REQUEST_SCOPE=2;
public static final int SESSION_SCOPE=3;
public static final int APPLICATION_SCOPE=4;

Ex: <%
pageContext.setAttribute("a", "aaa", pageConext.REQUEST_SCOPE);

%>

To get an attribute from page scope we have to use the following method.

public Object getAttribute(String name)

Ex: String a=(String)pageContext.getAttribute("a");

If we want to get an attribute value from the specified scope we have to use the following method.

public Object getAttribute(String name, int scope)

Ex: String uname=(String)pageContext.getAttribute("uname", pageContext.SESSION_SCOPE);

To remove an attribute from a particular we have to use the following method.

public void removeAttribute(String name, int scope)

Ex: pageContext.removeAttribute("a", pageContext.SESSION_SCOPE);

To find an attribute value from page scope, request scope, session scope and application scope we have to use the following method.

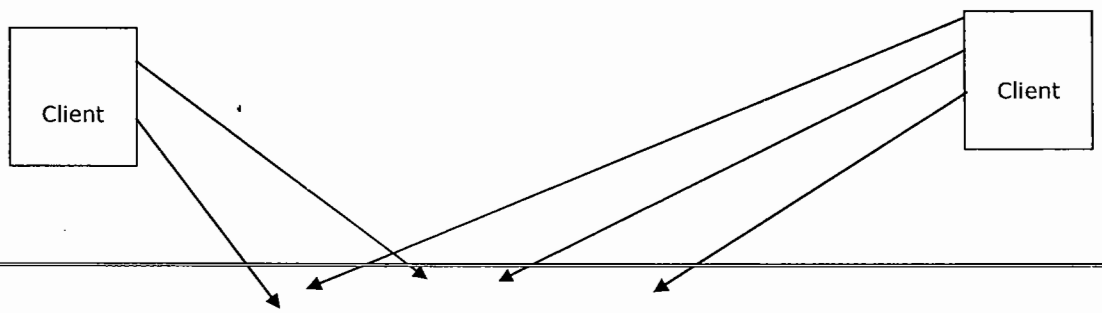
Public Object findAttribute(String name)

Ex: String name=pageContext.findAttribute("uname");

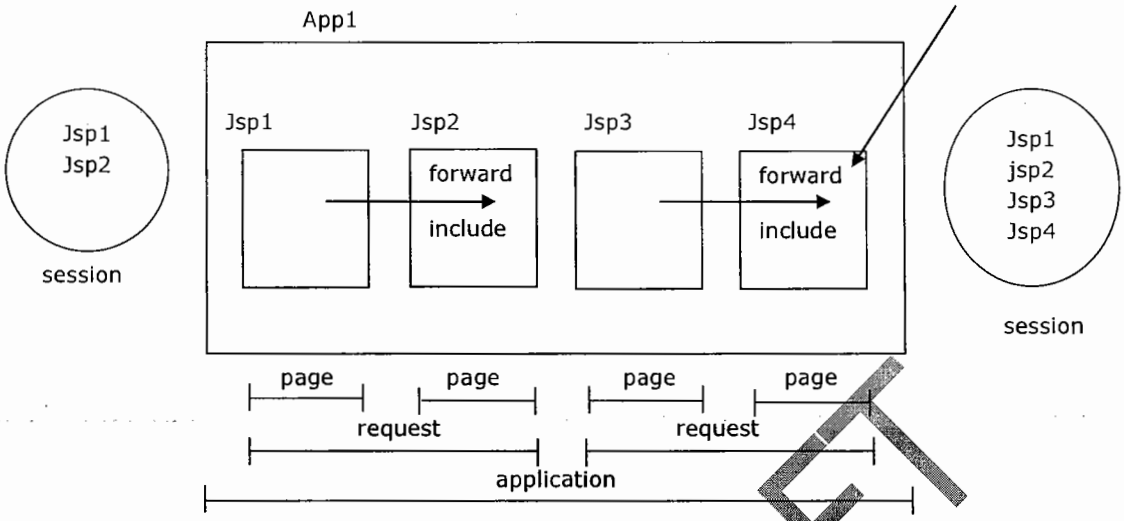
Jsp Scopes:

In J2SE applications, to define data availability i.e. scope for we have to use access specifiers public, protected, default and private.

Similarly to make available data to number of resources Jsp technology has provided the following 4 types of scopes with the access modifiers.



20



1. Page Scope:

If we declare the data in page scope by using pageContext object then that data should have the scope upto the present Jsp page.

2. Request Scope:

If we declare the data in request object then that data should have the scope up to the number of resources which are visited by the present request object.

3. Session Scope:

If we declare the data in HttpSession object then that data should have the scope up to the number of resources which are visited by the present client.

4. Application Scope:

If we declare the data in ServletContext object then that data should have the scope up to the number of resources which are available in the present web application.

----------Application1----------

App1:

employeedetails.html:

```
<html>
	<body bgcolor="lightblue"><br><br><br><br>
	<center><h1>Employee Details Form</h1></center>
	<pre><h2>
<form method="get" action="display.jsp">
		Employee Id :<input type="text" name="eid"/>
		Employee Name : <input type="text" name="ename"/>
		Employee Salary : <input type="text" name="esal"/>
		<input type="submit" value="Display"/>
	</h2></pre>
	</body>
</html>
```

display.jsp:

```
<%!
	int eid;
	String ename;
	float esal;
%>
<%
	try {
		eid=Integer.parseInt(request.getParameter("eid"));
		String ename=request.getParameter("ename");
		float esal=Float.parseFloat(request.getParameter("esal"));
	}
	catch(Exception e){
		e.printStackTrace();
	}
 %>
<html>
	<body>
		<center><h1>Employee Details</h1></center>
		<center>
		    Employee Id : <%=eid %><br><br>
		    Employee Name : <%=ename %><br><br>
		    Employee Salary : <%=esal %><br><br>
		</center>
	</body>
</html>
```


3. Jsp Actions:

In Jsp technology, by using scripting elements we are able to provide java code inside the Jsp pages, but the main theme of Jsp technology is not to allow java code inside Jsp pages.

To eliminate java code from Jsp pages we have to eliminate scripting elements, to eliminate scripting elements from Jsp pages we have to provide an alternative i.e. Jsp Actions.

In case of Jsp Actions, we will define a scripting tag in Jsp page and we will provide a block of java code w.r.t. scripting tag.

When container encounters the scripting tag then container will execute respective java code, by this an action will be performed called as Jsp Action.

In Jsp technology, there are 2 types of actions.

1. Standard Actions
2. Custom Actions

1. Standard Actions:

Standard Actions are Jsp Actions, which could be defined by the Jsp technology to perform a particular action.

Jsp technology has provided all the standard actions in the form of a set of predefined tags called Action Tags.

1. <jsp:useBean---->
2. <jsp:setProperty---->
3. <jsp:getProperty---->
4. <jsp:include---->
5. <jsp:forward---->
6. <jsp:param---->
7. <jsp:plugin---->
8. <jsp:fallback---->
9. <jsp:params---->
10. <jsp:declaration---->
11. <jsp:scriptlet---->
12. <jsp:expression---->

Java Beans:

Java Bean is a reusable component.

Java Bean is a normal java class which may declare properties, setter and getter methods in order to represent a particular user form at server side.

If we want to prepare Java Bean components then we have to use the following rules and regulations.

1. Java Bean is a normal java class, it is suggestible to implement Serializable interface.
2. Always Java Bean classes should be public, non-abstract and non-final.
3. In Java Bean classes, we have to declare all theproperties w.r.t. the properties define in the respective user form.
4. In Java Bean classes, all the properties should be private.
5. In Java Bean classes, all the behaviours should be public.
6. If we want to declare any constructor in Java Bean class then that constructor should be public and zero argument.

```
Ex: public class Employee implements Serializable {
        private String eno;
        private String ename;
        private float esal;
public void setEno(String eno) {
            this.eno=eno;
        }
        public void setEname(String ename) {
this.ename=ename;
        }
public void setEsal(String esal) {
this.esal=esal;
        }
public String getEno() {
            return eno;
        }
public String getEname() {
            return ename;
        }
public float getEsal() {
            return esal;
        }
    }
```

1. <jsp:useBean>:

The main purpose of <jsp:useBean> tag is to interact with bean object from a particular Jsp page.

Syntax:<jsp:useBean id="--" class="--" type="--" scope="--"/>

Where id attribute will take a variable to manage generated Bean object reference.

Where class attribute will take the fully qualified name of Bean class.

Where type attribute will take the fully qualified name of Bean class to define the type of variable in order to manage Bean object reference.

Where scope attribute will take either of the Jsp scopes to Bean object.

Note: In <jsp:useBean> tag, always it is suggestible to provide either application or session scope to the scope attribute value.

Ex: <jsp:useBean id="e" class="Employee" type="Employee" scope="session"/>

When container encounters the above tag then container will pick up class attribute value i.e. fully qualified name of Bean class then container will recognize Bean class .class file and perform Bean class loading and instantiation.

After creating Bean object container will assign Bean object reference to the variable specified as value to id attribute.

After getting Bean object reference container will store Bean object in a scope specified as value to scope attribute.

2. <jsp:setProperty>:

The main purpose of <jsp:setProperty> tag is to execute a particular setter method in order to set a value to a particular Bean property.

Syntax:<jsp:setProperty name="--" property="--" value="--"/>

Where name attribute will take a variable which is same as id attribute value in <jsp:useBean> tag.

Where property attribute will take a property name in order to access the respective setter method.

Where value attribute will take a value to pass as a parameter to the respective setter method.

3. <jsp:getProperty>:

The main purpose of <jsp:getProperty> tag is to execute a getter method in order to get a value from Bean object.

Syntax:<jsp:getProperty name="--" property="--"/>

Where name attribute will take a variable which is same as id attribute value in <jsp:useBean> tag.

Where property attribute will take a particular property to execute the respective getter method.

----------Application2----------

usebeanapp:

empform.html:

```
<html>
	<body bgcolor="lightblue"><br><br><br><br>
	<center><h1>Employee Details Form</h1></center>
	<form method="get" action="display.jsp">
	<pre><h2>
		Employee Id :<input type="text" name="eid"/>
		Employee Name : <input type="text" name="ename"/>
		Employee Salary : <input type="text" name="esal"/>
		<input type="submit" value="Display"/>
	</h2></pre>
	</form>
	</body>
</html>
```

Employee.java:

```
package comm.dss;
publicclass Employee implements java.io.Serializable {
	privateint eid;
	private String ename;
	privatefloat esal;
	publicint getEid() {
		return eid;
	}
	publicvoid setEid(int eid) {
		this.eid = eid;
	}
	public String getEname() {
		return ename;
	}
	publicvoid setEname(String ename) {
		this.ename = ename;
	}
	publicfloat getEsal() {
		return esal;
	}
	publicvoid setEsal(float esal) {
		this.esal = esal;
	}
}
```

display.jsp:

```
<%!
	int eid;
	String ename;
	float esal;
%>
```

```
<%
	try {
		eid=Integer.parseInt(request.getParameter("eid"));
		ename=request.getParameter("ename");
		esal=Float.parseFloat(request.getParameter("esal"));
	}
	catch(Exception e){
		e.printStackTrace();
	}
 %>
<jsp:useBean id="e" class="com.dss.EmployeeBean" type="com.dss.EmployeeBean"
scope="session">
<jsp:setProperty name="e" property="eid" value='<%=eid %>'/>
<jsp:setProperty name="e" property="ename" value='<%=ename %>'/>
<jsp:setProperty name="e" property="esal" value='<%=esal %>'/>
<html>
	<body>
		<center><h1>Employee Details</h1></center>
		<center>
			Employee Id : <jsp:getProperty name="e"
property="eid"/><br><br>
			Employee Name : <jsp:getProperty name="e"
property="ename"/><br><br>
			Employee Salary : <jsp:getProperty name="e"
property="esal"/><br><br>
		</center>
	</body>
</html>
</jsp:useBean>
```

Note: In case of <jsp:useBean> tag, in general we will provide a separate <jsp:setProperty> tag to set a particular value to the respective property in Bean object.

In case of <jsp:useBean> tag, it is possible to copy all the request parameter values directly onto the respective Bean object.

To achieve this we have to provide "*" as value to property attribute in <jsp:setProperty> tag.

Ex: <jsp:setProperty name="e" property="*"/>

If we want to achieve the above requirement then we have to maintain same names in the request parameters i.e. form properties and Bean properties.

Note: The above "*" notation is not possible with <jsp:getProperty> tag.

4. <jsp:include>:

Q: What are the differences between include directive and <jsp:include> action tag?

Ans: 1. In Jsp applications, **include directive** can be used to include the content of the target resource into the present Jsp page.

In Jsp pages, **<jsp:include>** action tag can be used to include the target resource response into the present Jsp page response.

2. In general include directive can be used to include static resources where the frequent updations are not available.

<jsp:include> action tag can be used to include dynamic resources where the frequent updations are available.

3. In general directives will be resolved at the time of translation and actions will be resolved at the time of request processing. Due to this reason include directive will be resolved at the time of translation but <jsp:include> action tag will be resolved at the time of request processing.

If we are trying to include a target Jsp page into present Jsp page by using include directive then container will prepare only one translated servlet.

To include a target Jsp page into the present Jsp page if we use <jsp:include> action tag then container will prepare 2 separate translated servlets.

In Jsp applications, include directives will provide static inclusion, but <jsp:include> action tag will provide dynamic inclusion.

In Jsp technology, <jsp:include> action tag was designed on the basis of Include Request Dispatching Mechanism.

Syntax:<jsp:include page="--" flush="--"/>

Where page attribute will take the name and location of the target resource to include its response.

Where flush is a boolean attribute, it can be used to give an intimation to the container about to autoFlush or not to autoFlush dynamic response to client when JspWriter buffer filled with the response at the time of including the target resource response into the present Jsp page.

----------Application3----------

includeapp:

addform.html:

```
<html>
	<body bgcolor="lightgreen">
	<form action="add.jsp">
		<pre>
			<u>Product Details</u>
			Product Id : <input type="text" name="pid"/>
			Product Name : <input type="text" name="pname"/>
			Product Cost : <input type="text" name="pcost"/>
			<input type="submit" value="ADD"/>
```

```
            </pre>
        </form>
        </body>
</html>
```

add.jsp:

```
<%@page import="java.sql.*"%>
<%!
        String pid;
        String pname;
        int pcost;
        static Connection con;
        static Statement st;
        ResultSet rs;
        ResultSetMetaData rsmd;
        static{
                try{
                        Class.forName("oracle.jdbc.driver.OracleDriver");
        con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system",
"durga");
                        st=con.createStatement();
                }
                catch(Exception e){
                        e.printStackTrace();
                }
        }
%>
<%
        try{
                pid=request.getParameter("pid");
                pname=request.getParameter("pname");
                pcost=Integer.parseInt(request.getParameter("pcost"));
                st.executeUpdate("insert into product
values('"+pid+"','"+pname+"',"+pcost+")");
                rs=st.executeQuery("select * from product");
                rsmd=rs.getMetaData();
                int count=rsmd.getColumnCount();
%>
        <html><body><center>
        <table border="1" bgcolor="lightyellow">
        <tr>
<%
        for (int i=1;i<=count;i++){
%>
        <td><b><font size="6" color="red">
        <center><%=rsmd.getColumnName(i) %></center>
        </font></b></td>
<%
        }
%>
        </tr>
<%
```

```
        while(rs.next()){
%>
        <tr>
<%
        for(int i=1;i<=count;i++){
%>
        <td><b><font size="6">
<%=rs.getString(i) %>
        </font></b></td>
<%
        }
%>
        </tr>
<%
        }
%>
        </table></center></body></html>
<%
        }
        catch(Exception e){
                e.printStackTrace();
        }
%>
        <hr>
        <jsp:include page="addform.html" flush="true"/>
```


5. <jsp:forward>:

Q: What are the differences between<jsp:include> action tag and <jsp:forward> action tag?

Ans: 1. **<jsp:include>** action tag can be used to include the target resource response into the present Jsp page.

<jsp:forward> tag can be used to forward request from present Jsp page to the target resource.

2. <jsp:include> tag was designed on the basis of Include Request Dispatching Mechanism.

<jsp:forward> tag was designed on the basis of Forward Request Dispatching Mechanism.

3. When Jsp container encounter <jsp:include> tag then container will forward request to the target resource, by executing the target resource some response will be generated in the response object, at the end of the target resource container will bypass request and response objects back to first resource, at the end of first resource execution container will

dispatch overall response to client. Therefore, in case of <jsp:include> tag client is able to receive all the resources response which are participated in the present request processing.

When container encounters<jsp:forward> tag then container will bypass request and response objects to the target resource by refreshing response object i.e. by eliminating previous response available in response object, at the end of target resource container will dispatch the generated dynamic response directly to the client without moving back to first resource. Therefore, in case of <jsp:forward> tag client is able to receive only target resource response.

Syntax:<jsp:forward page="--"/>

Where page attribute specifies the name and location of the target resource.

6. <jsp:param>:

This action tag can be used to provide a name value pair to the request object at the time of by passing request object from present Jsp page to target page either in include mechanism or in forward mechanism or in both.

This tag must be utilized as a child tag to <jsp:include> tag and <jsp:forward> tags.

Syntax:<jsp:param name="--" value="--"/>

----------Application4----------

forwardapp:

registrationform.html:

```
<html>
	<body bgcolor="lightgreen">
	<form action="registration.jsp">
		<pre>
			<u>Registration Form</u>
			Name : <input type="text" name="pname"/>
			Age : <input type="text" name="uage"/>
			Address : <input type="text" name="uaddr"/>
			<input type="submit" value="Registration"/>
		</pre>
	</form>
	</body>
</html>
```

existed.jsp:

```
<center><h1>User Existed</h1></center>
```

success.jsp:

```
<center><h1>Registration Success</h1></center>
```

failure.jsp:

```
<center><h1>Registration Failure</h1></center>
```

registration.jsp:

```
<%@page import="java.sql.*"%>
<%!
	String uname;
	int uage;
	String uaddr;
	static Connection con;
	static Statement st;
	ResultSet rs;
	static{
		try{
			Class.forName("oracle.jdbc.driver.OracleDriver");

	con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system",
"durga");
			st=con.createStatement();
		}
		catch(Exception e){
			e.printStackTrace();
		}
	}
%>
<%
	try{
		uname=request.getParameter("uname");
		uage=Integer.parseInt(request.getParameter("uage"));
		uaddr=request.getParameter("uaddr");
		rs=st.executeQuery("select * from reg_users where uname='"+uname+"'");
		boolean b=rs.next();
		if(b==true)
		{
%>
	<jsp:forward page="existed.jsp"/>
<%
		}
		else
		{
			int rowCount=st.executeUpdate("insert into reg_users values
('"+uname+"',"+uage+",'"+uaddr+"')");
			if(rowCount == 1)
			{
%>
	<jsp:forward page="success.jsp"/>
<%
			}
			else
			{
```

```
%>
        <jsp:forward page="failure.jsp"/>
<%
                        }
                }
        }
        catch(Exception e){
%>
        <jsp:forward page="failure.jsp"/>
<%
                e.printStackTrace();
        }
%>
```

7. <jsp:plugin>:

This tag can be used to include an applet into the present Jsp page.

Syntax:<jsp:plugin code="--" width="--" height="--" type="--"/>

Where code attribute will take fully qualified name of the applet.

Where width and height attributes can be used to specify the size of applet.

Where type attribute can be used to specify which one we are going to include whether it is applet or bean.

Ex:<jsp:plugin code="Logo" width="1000" height="150" type="applet"/>

8. <jsp:params>:

In case of the applet applications, we are able to provide some parameters to the applet in order to provide input data.

Similarly if we want to provide input parameters to the applet from <jsp:plugin> tag we have to use <jsp:param> tag.
<jsp:param> tag must be utilized as a child tag to <jsp:params> tag.
<jsp:params> tag must be utilized as a child tag to <jsp:plugin> tag.

Syntax:

```
<jsp:plugin>
<jsp:params>
<jsp:param name="--" value="--"/>
<jsp:param name="--" value="--"/>
          ----------
</jsp:params>
    ----------
</jsp:plugin>
```

If we provide any input parameter to the applet then that parameter value we are able to get by using the following method from Applet class.

public String getParameter(String name)

Ex: String msg=getParameter("message");

----------Application5----------

pluginapp:

LogoApplet.java:

```
import java.awt.*;
import java.applet.*;
public class LogoApplet extends Applet
{
        String msg;
        public void paint(Graphics g)
        {
        msg=getParameter("message");
        Font f=new Font("arial",Font.BOLD,40);
        g.setFont(f);
        this.setBackground(Color.blue);
        this.setForeground(Color.white);
        g.drawString(msg,150,70);
        }
}
```

logo.jsp:

```
<jsp:plugin code="LogoApplet" width="1000" height="150" type="applet">
        <jsp:params>
        <jsp:param name="message" value="durga software solutions"/>
        </jsp:params>
</jsp:plugin>
```


9. <jsp:fallback>:

The main purpose of <jsp:fallback> tag is to display an alternative message when client browser is not supporting <OBJECT---> tag and <EMBED---> tag.

Syntax:<jsp:fallback>-------Description---------</jsp:fallback>

In Jsp applications, we have to utilize <jsp:fallback> tag as a child tag to <jsp:plugin> tag.

Ex:<jsp:plugin code="LogoApplet" width="1000" height="150" type="applet">

<jsp:fallback>Applet Not Allowed</jsp:fallback>

</jsp:plugin>

10. <jsp:declaration>:

This tag is almost all same as the declaration scripting element, it can be used to provide all the Java declarations in the present Jsp page.

Syntax:<jsp:declaration>

-------- } Java Declarations

</jsp:declaration>

11. <jsp:scriptlet>:

This tag is almost all same as thescripting element scriptlets, it can be used to provide a block of Java code in Jsp pages.

Syntax:<jsp:scriptlet>

-------- } Block of Java code

</jsp:scriptlet>

12. <jsp:expression>:

This tag is almost all same as the scripting element expression, it can be used to provide a Java expression in the present Jsp page.

Syntax:<jsp:expression> Java Expression </jsp:expression>

Ex:

```
<%@page import="java.util.*"%>
<jsp:declaration>
Date d=null;
String date=null;
</jsp:declaration>
<jsp:scriptlet>
d=new Date();
date=d.toString();
</jsp:scriptlet>
<html>
<body bgcolor="lightyellow">
<center><b><font size="6" color="red"><br><br>
Today Date : <jsp:expression>date</jsp:expression>
</font></b></center></body>
</html>
```

2. Custom Actions:

In Jsp technology, by using Jsp directives we are able to define present Jsp page characteristics, we are unable to use Jsp directives to perform actions in the Jsp pages.

To perform actions if we use scripting elements then we have to provide Java code inside the Jsp pages.

The main theme of Jsp technology is not to allow Java code inside Jsp pages, to eliminate Java code from Jsp pages we have to eliminate scripting elements.

If we want to eliminate scripting elements from Jsp pages we have to use actions.

There are 2 types of actions in Jsp technology.

1. Standard Actions
2. Custom Actions

Standard Actions are predefined actions provided by Jsp technology, these standard actions are limited in number and having bounded functionalities so that standard actions are not sufficient to satisfy the complete client requirement.

In this context, there may be a requirement to provide Java code inside the Jsp pages so that to eliminate Java code completely from Jsp pages we have to use Custom Actions.

Custom Actions are Jsp Actions which could be prepared by the developers as per their application requirements.

In Jsp technology, standard actions will be represented in the form of a set of predefined tags are called as **Action Tags.**

Similarly all the custom actions will be represented in the form of a set of user defined tags are called as **Custom Tags.**

To prepare custom tags in Jsp pages we have to use the following syntax.

Syntax:<prefix_Nmae:tag_Name>

---} Body

</prefix_Name>

If we want to design custom tags in our Jsp applications then we have to use the following 3 elements.

1. Jsp page with taglib directive
2. TLD(Tag Library Descriptor) file
3. TagHandler class

Where TagHandler class is a normal Java class it is able to provide the basic functionality for the custom tags.

Where TLD file is a file, it will provide the mapping between custom tag names and respective TagHandler classes.

Where taglib directive in the Jsp page can be used to make available the tld files into the present Jsp page on the basis of custom tags prefix names.

Internal Flow:

When container encounters a custom tag container will pick up the custom tag name and the respective prefix name then recognize a particular taglib directive on the basis of the prefix attribute value.

After recognizing taglib directive container will pick up uri attribute value i.e. the name and location of tld file then container will recognize the respective tld file.

After getting tld file container will identify the name and location of TagHandler class on the basis of custom tag name.

When container recognize the respective TagHandler class .class file then container will perform TagHandler class loading, instantiation and execute all the life cycle methods.

1. Taglib Directive:

In custom tags design, the main purpose of taglib directive is to make available the required tld file into the present Jsp page and to define prefix names to the custom tags.

Syntax:<%@taglib uri="--" prefix="--"%>

Where prefix attribute can be used to specify a prefix name to the custom tag, it will have page scope i.e. the specified prefix name is valid up to the present Jsp page.

Where uri attribute will take the name and location of the respective tld file.

2. TLD File:

The main purpose of TLD file is to provide the mapping between custom tag names and the respective TagHandler classes and it is able to manage the description of the custom tags attributes.

To provide the mapping between custom tag names and the respective TagHandler class we have to use the following tags.

```
<taglib>
<jsp-version>jsp version</jsp-version>
<tlib-version>tld file version</tlib-version>
<short-name>tld file short name</short-name>
<description>description about tld file</description>
```

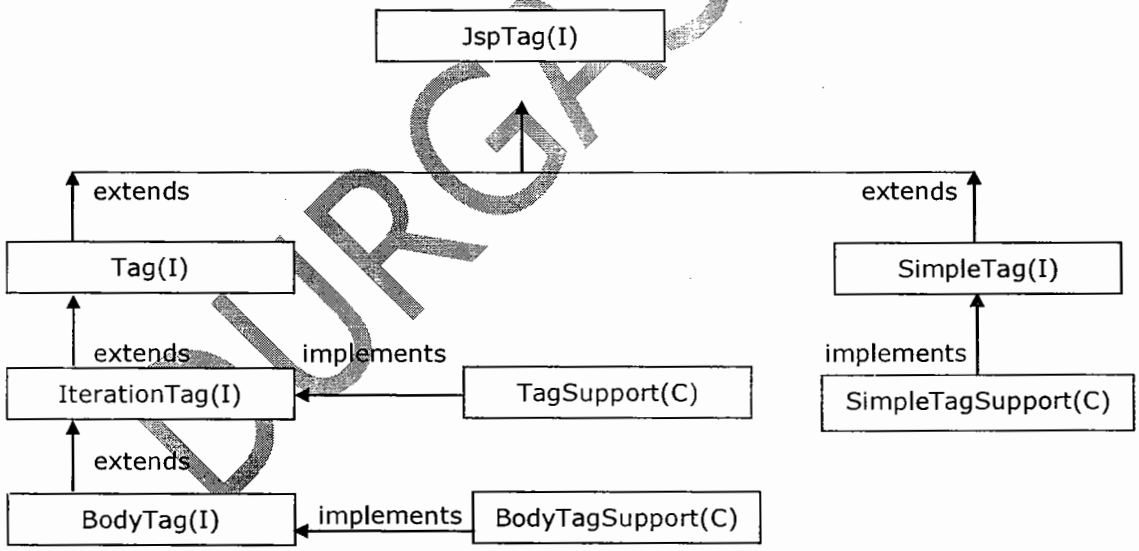
```
<tag>
<name>custom tag name</name>
<tag-class>fully qualified name of TagHandler class</tag-class>
<body-content>jsp or empty</body-content>
<short-name>custom tag short name</short-name>
<description>description about custom tags</description>
</tag>
        ---------
</taglib>
```

3. TagHandler class:

In custom tags preparation, the main purpose of TagHandler class is to define the basic functionality for the custom tags.

To design TagHandler classes in custom tags preparation Jsp API has provided some predefined library in the form of javax.servlet.jsp.tagext package (tagext-->tag extension).

javax.servlet.jsp.tagext package has provided the following library to design TagHandler classes.



The above Tag library was divided into 2 types.

1. Classic Tag library
2. Simple Tag library

As per the tag library provided by Jsp technology there are 2 types of custom tags.

1. Classic tags

2. Simple Tags

1. Classic Tags:

Classic Tags are the custom tags will be designed on the basis of Classic tag library provided by Jsp API.

As per the classic tag library provided by Jsp API there are 3 types of classic tags.

1. Simple Classic Tags
2. Iterator Tags
3. Body Tags

2. Simple Classic Tags:

Simple Classic Tags are the classic tags, which should not have body and attributes list.

To design simple classic tags the respective TagHandler class must implement Tag interface either directly or indirectly.

```
public interface Tag extends JspTag
{
    public static final int EVAL_BODY_INCLUDE;
    public static final int SKIP_BODY;
    public static final int EVAL_PAGE;
    public static final int SKIP_PAGE;
    public void setPageContext(PageContext pageContext);
    public void setParent(Tag t);
    public Tag getParent();
    public int doStartTag()throws JspException;
    public int doEndTag()throws JspException;
    public void release();
}
public class MyHandler implements Tag
{
    --------
    --------
}
```


Where the purpose of setPageContext(_) method is to inject pageContext implicit object into the present TagHandler class.

Where the purpose of setParent(_) method is to inject parent tags TagHandler class object into the present TagHandler class.

Where the purpose of getParent() method is to return the parent tags TagHandler class object from the TagHandler class.

Where the purpose of doStartTag() method is to perform a particular action when container encounters the start tag of the custom tag.

After the custom tags start tag evaluating the custom tag body is completely depending on the return value provided by doStartTag () method.

There are 2 possible return values from doStartTag() method.

1. EVAL_BODY_INCLUDE
2. SKIP_BODY

If doStartTag() method returns EVAL_BODY_INCLUDE constant then container will evaluate the custom tag body.

If doStartTag() method returns SKIP_BODY constant then container will skip the custom tag body and encounter end tag.

Where the purpose of doEndTag() method is to perform an action when container encounters end tag of the custom tag.

Evaluating the remaining the Jsp page after the custom tag or not is completely depending on the return value provided by doEndTag() method.

There are 2 possible return values from doEndTag() method.

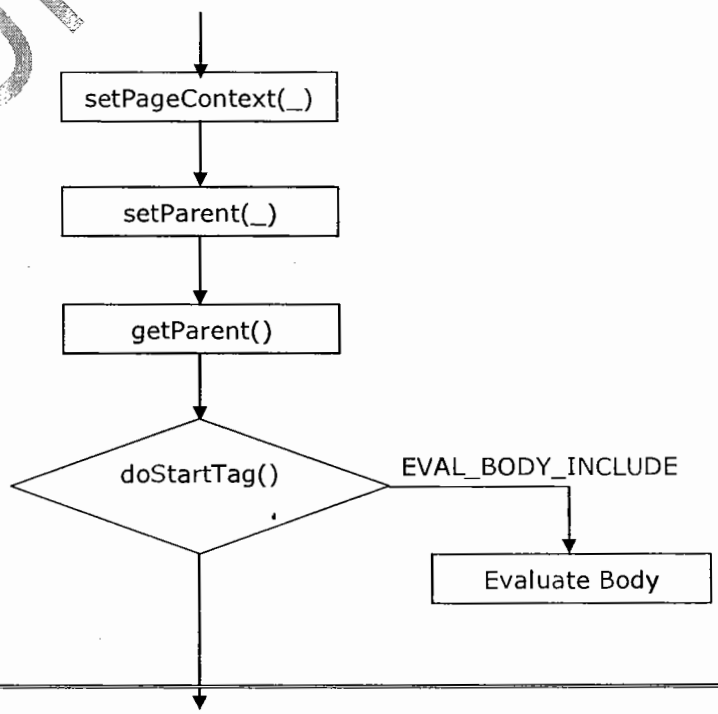
1. EVAL_PAGE
2. SKIP_PAGE

If doEndTag() method returns EVAL_PAGE constant then container will evaluate the remaining Jsp page.

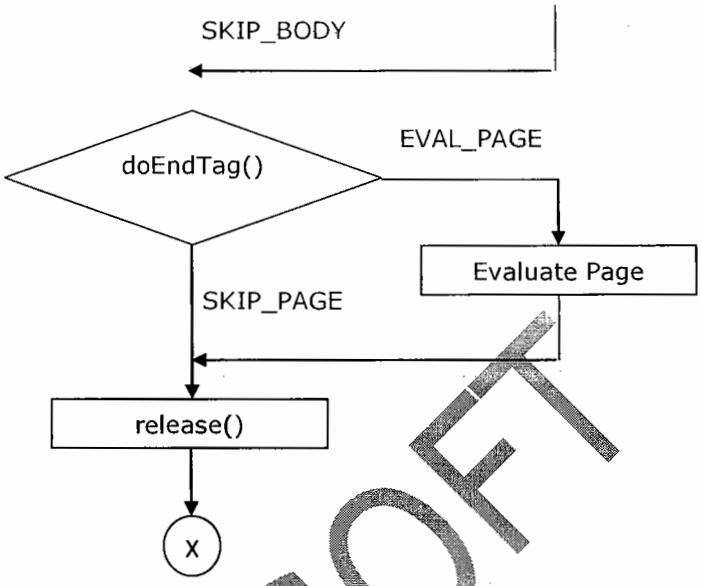
If doEndTag() method returns SKIP_PAGE constant then container will not evaluate the remaining Jsp page.

Where release() method can be used to perform TagHandler class deinstantiation.

Life Cycle of Tag interface:



40



Note: In Tag interface life cycle, container will execute getParent() method when the present custom tag is child tag to a particular parent tag otherwise container will not execute getParent() method.

----------Application6----------

custapp1:

hello.jsp:

```
<%@taglib uri="/WEB-INF/hello.tld" prefix="mytags"%>
<mytags:hello/>
```

hello.tld:

```
<taglib>
        <jsp-version>2.1</jsp-version>
        <tlib-version>1.0</tlib-version>
        <tag>
                <name>hello</name>
                <tag-class>com.dss.HelloHandler</tag-class>
                <body-content>jsp</body-content>
        </tag>
</taglib>
```

HelloHandler.java:

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class HelloHandler implements Tag
{
	PageContext pageContext;
	public void setPageContext(PageContext pageContext)
	{
		this.pageContext=pageContext;
		System.out.println("setPageContext()");
	}
	public void setParent(Tag t)
	{
		System.out.println("setParent()");
	}
	public Tag getParent()
	{
		System.out.println("getParent()");
		return null;
	}
	public int doStartTag() throws JspException
	{
		try
		{
			System.out.println("doStartTag()");
			JspWriter out=pageContext.getOut();
			out.println("<h1>Hello..... First Custom Tag Application</h1>");
		}
		catch (Exception e)
		{
			e.printStackTrace();
		}
		return SKIP_BODY;
	}
	public int doEndTag()throws JspException
	{
		System.out.println("doEndTag()");
		return SKIP_PAGE;
	}
	public void release()
	{}
}
```


Note: To compile above code we need to set the classpath environment variable to the location of jsp-api.jar file.

Observations:

Case 1: In the above application, if we provide <body-content> type is empty in the tld file and if we provide body to the custom tag then container will raise an Exception like

org.apache.jasper.JasperException:/hello.jsp(2,0) According to TLD, tag mytags:hello must be empty, but is not.

Case 2: If we provide<body-content> value as jsp in tld file, if we provide body to custom tag in jsp page page and if we return SKIP_BODY constant in the respective TagHandler class then container won't raise any Exception but container won't evaluate custom tag body.

Attributes in Custom Tgas:

If we want to provide attributes in custom tags then we have to perform the following steps.

Step 1: Define attribute in the custom tag.

Ex:<mytags:hello name="Durga"/>

Step 2: Provide attributes description in the respective tld file.

To provide attributes description in tld file we have to use the following tags in tld file.

```
<taglib>
      --------
<tag>
          ------------
<attribute>
<name>attribute_name</name>
<required>true/false</required>
<rtexprvalue>true/false</rtexprvalue>
</attribute>
</tag>
</taglib>
```

Where <attribute> tag can be used to represent a single attribute in the tld file.
Where <name> tag will take attribute name.

Where <required> tag is a boolean tag, it can be used to specify whether the attribute is mandatory or optional.

Where <rtexprvalue> tag can be used to specify whether the attribute accept runtime values or not.

Step 3: Declare a property and setter method in TagHandler class with the same name of the attribute defined in custom tag.

```
public class MyHandler implements Tag {
   private String name;
   public void setName(String name) {
      this.name=name;
   }
```

```
    ----------
}
```

2. Iterator Tags:

Iterator tags are the custom tags, it will allow to evaluate custom tag body repeatedly.

If we want to prepare iterator tags the respective TagHandler class must implement javax.servlet.jsp.tagext.IterationTag interface.

```
public interface IterationTag extends Tag {
    public static final int EVAL_BODY_INCLUDE;

    public static final int SKIP_BODY;

    public static final int EVAL_PAGE;

    public static final int SKIP_PAGE;

    public static final int EVAL_BODY_AGAIN;

    public void setPageContext(PageContext pageContext);

    public void setParent(Tag t);

    public Tag getParent();

    public int doStartTag()throws JspException;

    public int doAfterBody()throws JspException;

    public int doEndTag()throws JspException;

    public void release();
}

public class MyHandler implements IterationTag
{ ----- }
```


In general there are 2 possible return values from doStartTag() method.

1. EVAL_BODY_INCLUDE
2. SKIP_BODY

If we return SKIP_BODY constant from doStartTag() method then container will skip the custom tag body.

If we return EVAL_BODY_INCLUDE constant from doStartTag() method then container will execute the custom tag body.

Note: In case of iterator tags, we must return EVAL_BODY_INCLUDE from doStartTag() method.

After evaluating the custom tag body in case of iterator tags, container will access doAfterBody() method.

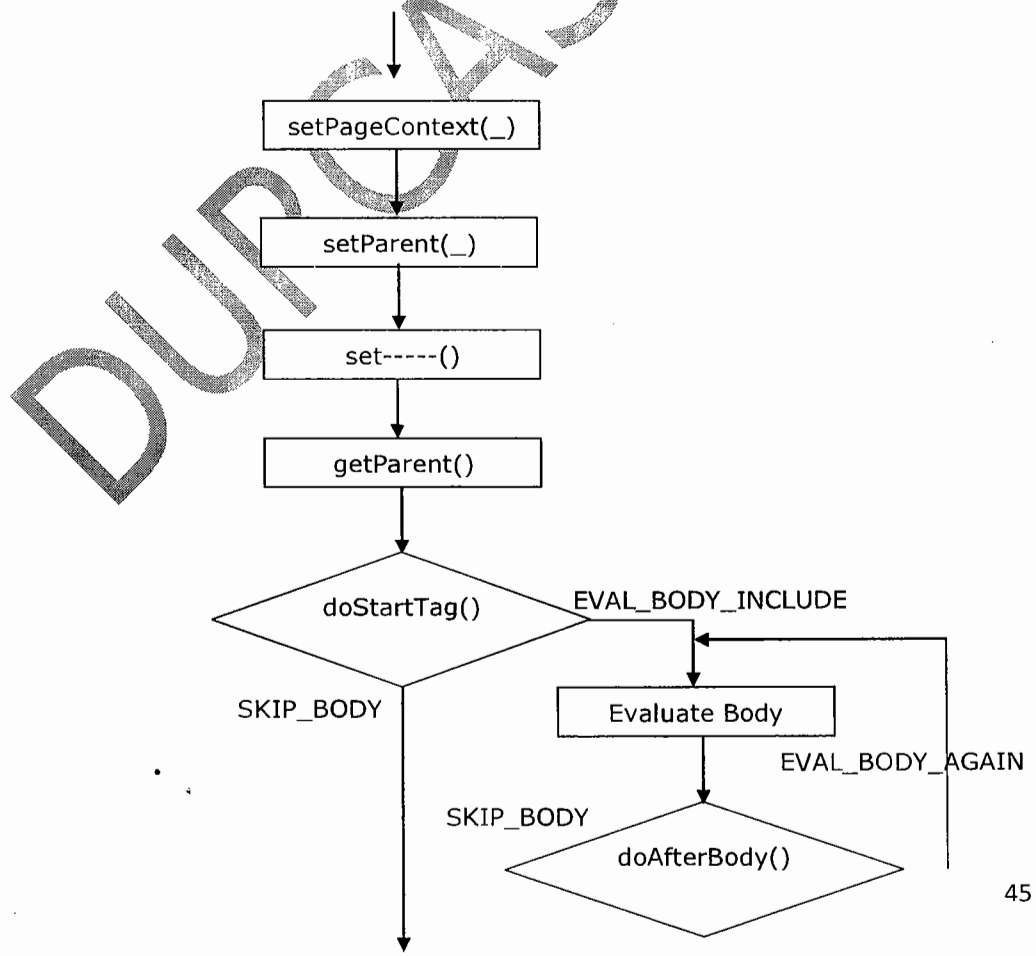
In the above context, evaluating the custom tag body again or not is completely depending on the return value which we are going to return from doAfterBody() method.

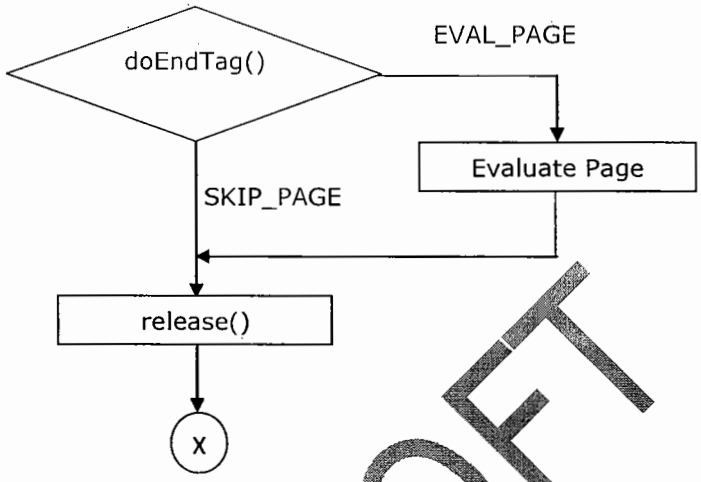
1. EVAL_BODY_AGAIN
2. SKIP_BODY

If we return EVAL_BODY_AGAIN constant from doAfterBody() method then container will execute the custom tag body again.

If we return SKIP_BODY constant from doAfterBody() method then container will skip out custom tag body evaluation and encounter end tag of the custom tag.

Life Cycle of IterationTag interface:





If we want to design custom tagsby using above approach then the respective TagHandler class must implement Tag interface and IterationTag interface i.e. we must provide the implementation for all the methods which are declared in Tag and IterationTag interfaces in our TagHandler class.

This approach will increase burden to the developers and unnecessary methods in TgaHandler classes.

To overcome the above problem Jsp API has provided an alternative in the form of TagSupport class.

TagSupport is a concrete class, which was implemented Tag and IterationTag interfaces with the default implementation.

If we want to prepare custom tags with the TagSupport class then we have to take an user defined class, which must be a subclass to TagSupport class.

```
public interface TagSupport implements IterationTag {
    public static final int EVAL_BODY_INCLUDE;
    public static final int SKIP_BODY;
    public static final int EVAL_PAGE;
    public static final int SKIP_PAGE;
    public static final int EVAL_BODY_AGAIN;
    public PageContext pageContext;
    public Tag t;
    public void setPageContext(PageContext pageContext) {
        this.pageContext=pageContext;
    }
    public void setParent(Tag t) {
        this.t=t;
    public Tag getParent() {
        return t;
    }
```

```
    public int doStartTag()throws JspException {
        return SKIP_BODY;
    }
    public int doAfterBody()throws JspException {
        return SKIP_BODY;
    }
    public int doEndTag()throws JspException {
        return EVAL_PAGE;
    }
    public void release() { }
}
  public class MyHandler implements TagSupport
{ ----- }
```

----------Application7----------

custapp2:

iterate.jsp:

```
<%@taglib uri="/WEB-INF/iterate.tld" prefix="mytags"%>
<mytags:iterate times="10"><br>
        Durga Software Solutions
</mytags:iterate>
```

iterate.tld:

```
<taglib>
        <jsp-version>2.1</jsp-version>
        <tlib-version>1.0</tlib-version>
        <tag>
                <name>iterate</name>
                <tag-class>com.dss.Iterate</tag-class>
                <body-content>jsp</body-content>
                <attribute>
                        <name>times</name>
                        <required>true</required>
                        <rtexprvalue>true</rtexprvalue>
                </attribute>
        </tag>
</taglib>
```

Iteration.java:

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class Iterate extends TagSupport
{
        int count=1;
        private int times;
        public void setTimes(int times)
        {
```

```
            this.times=times;
        }
        public int doStartTag() throws JspException
        {
            return EVAL_BODY_INCLUDE;
        }
        public int doAfterBody() throws JspException
        {
            if(count<times)
            {
                count++;
                return EVAL_BODY_AGAIN;
            }
            else
                return SKIP_BODY;
        }
}
```

Nested Tags:

Defining a tag inside a tag is called as **Nested Tag.**

In custom tags application, if we declare any nested tag then we have to provide a separate configuration in tld file and we have to prepare a separate TagHandler class under classes folder.

----------Application8----------

custapp3:

nested.jsp:

```
<%@taglib uri="/WEB-INF/nested.tld" prefix="mytags"%>
<h1><center>
<mytags:if condition='<%=10>20%>'>
        <mytags:true>condition is true</mytags:true>
        <mytags:false>condition is false</mytags:false>
</mytags:if>
</center></h1>
```

nested.tld:

```
<taglib>
        <jsp-version>2.1</jsp-version>
        <tlib-version>1.0</tlib-version>
```

```
        <tag>
                <name>if</name>
                <tag-class>com.dss.If</tag-class>
                <body-content>jsp</body-content>
                <attribute>
                        <name>condition</name>
                        <required>true</required>
                        <rtexprvalue>true</rtexprvalue>
                </attribute>
        </tag>
        <tag>
                <name>true</name>
                <tag-class>com.dss.True</tag-class>
                <body-content>jsp</body-content>
        </tag>
        <tag>
                <name>false</name>
                <tag-class>com.dss.False</tag-class>
                <body-content>jsp</body-content>
        </tag>
</taglib>
```

If.java:

```
package com.dss;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class If extends TagSupport
{
        private boolean condition;
        public void setCondition(boolean condition)
        {
                this.condition=condition;
        }
        public boolean getCondition()
        {
                return condition;
        }
        public int doStartTag() throws JspException
        {
                return EVAL_BODY_INCLUDE;
        }
}
```

True.java:

```
package com.dss;
```

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class True extends TagSupport
{
	public int doStartTag() throws JspException
	{
		If i=(If)getParent();
		boolean condition=i.getCondition();
		if(condition == true)
			return EVAL_BODY_INCLUDE;
		else
			return SKIP_BODY;
	}
}
```

False.java:

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class False extends TagSupport
{
	public int doStartTag() throws JspException
	{
		If i=(If)getParent();
		boolean condition=i.getCondition();
		if(condition == true)
			return SKIP_BODY;
		else
			return EVAL_BODY_INCLUDE;
	}
}
```


----------Application9----------

custapp4:

empdetails.jsp:

```
<%@taglib uri="/WEB-INF/emp.tld" prefix="emp"%>
<emp:empDetails/>
```

emp.tld:

```
<taglib>
	<jsp-version>2.1</jsp-version>
	<tlib-version>1.0</tlib-version>
	<tag>
		<name>empDetails</name>
		<tag-class>com.dss.EmpDetails</tag-class>
```

```
            <body-content>empty</body-content>
        </tag>
</taglib>
```

EmpDetails.java:

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.sql.*;
public class EmpDetails extends TagSupport
{
        Connection con;
        Statement st;
        ResultSet rs;
        public EmpDetails()
        {
                try
                {
                        Class.forName("oracle.jdbc.driver.OracleDriver");
        con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","
durga");
                        st=con.createStatement();
                }
                catch (Exception e)
                {
                        e.printStackTrace();
                }
        }

public int doStartTag() throws JspException
        {
                try
                {
                        JspWriter out=pageContext.getOut();
                        rs=st.executeQuery("select * from emp");
                        ResultSetMetaData rsmd=rs.getMetaData();
                        int count=rsmd.getColumnCount();
                        out.println("<html><body bgcolor='pink'>");
                        out.println("<center><br><br>");
                        out.println("<table border='1' bgcolor='lightyellow'>");
                        out.println("<tr>");
                        for (int i=1;i<=count;i++)
                        {
                                out.println("<td><b><font size='6'
color='red'><center>"+rsmd.getColumnName(i)+"</center></font></b></td>");
                        }
                        out.println("</tr>");
                        while (rs.next())
                        {
                                out.println("<tr>");
                                for (int i=1;i<=count;i++)
                                {
```

```
                                    out.println("<td><b><font
size='5'>"+rs.getString(i)+"</font></b></td>");
                                }
                                out.println("</tr>");
                        }
                        out.println("</table></center></body></html>");
                }
                catch (Exception e)
                {
                        e.printStackTrace();
                }
                return SKIP_BODY;
        }
}
```

3. Body Tags:

Up to now in our custom tags (simple classic tags, iteration tags) we prepared custom tags with the body, where we did not perform updations over the custom tag body, just we scanned custom tag body and displayed on the client browser.

If we want to perform updations over the custom tag body then we have to use Body Tags.

If we want to design body tags in Jsp technology then the respective TagHandler class must implement BodyTag interface either directly or indirectly.

```
public interface BodyTag extends IterationTag
{
    public static final int EVAL_BODY_INCLUDE;
    public static final int SKIP_BODY;
    public static final int EVAL_PAGE;
    public static final int SKIP_PAGE;
    public static final int EVAL_BODY_AGAIN;
    public static final int EVAL_BODY_BUFFERED;
    public void setPageContext(PageContext pageContext);
    public void setParent(Tag t);
    public Tag getParent();
    public int doStartTag()throws JspException;
    public void doInitBody()throws JspException;
    public void setBodyContent(BodyContent bodyContent);
    public int doAfterBody()throws JspException;
    public int doEndTag()throws JspException;
    public void release();
}
public class MyHandler implements BodyTag
{ ----- }
```

In case of body tags, there are 3 possible return values from doStartTag() method.

1. EVAL_BODY INCLUDE

2. SKIP_BODY
3. EVAL_BODY_BUFFERED

If we return EVAL_BODY_INCLUDE constant from doStartTag() method then container will evaluate the custom tag body i.e. display as it is the custom tag body on client browser.

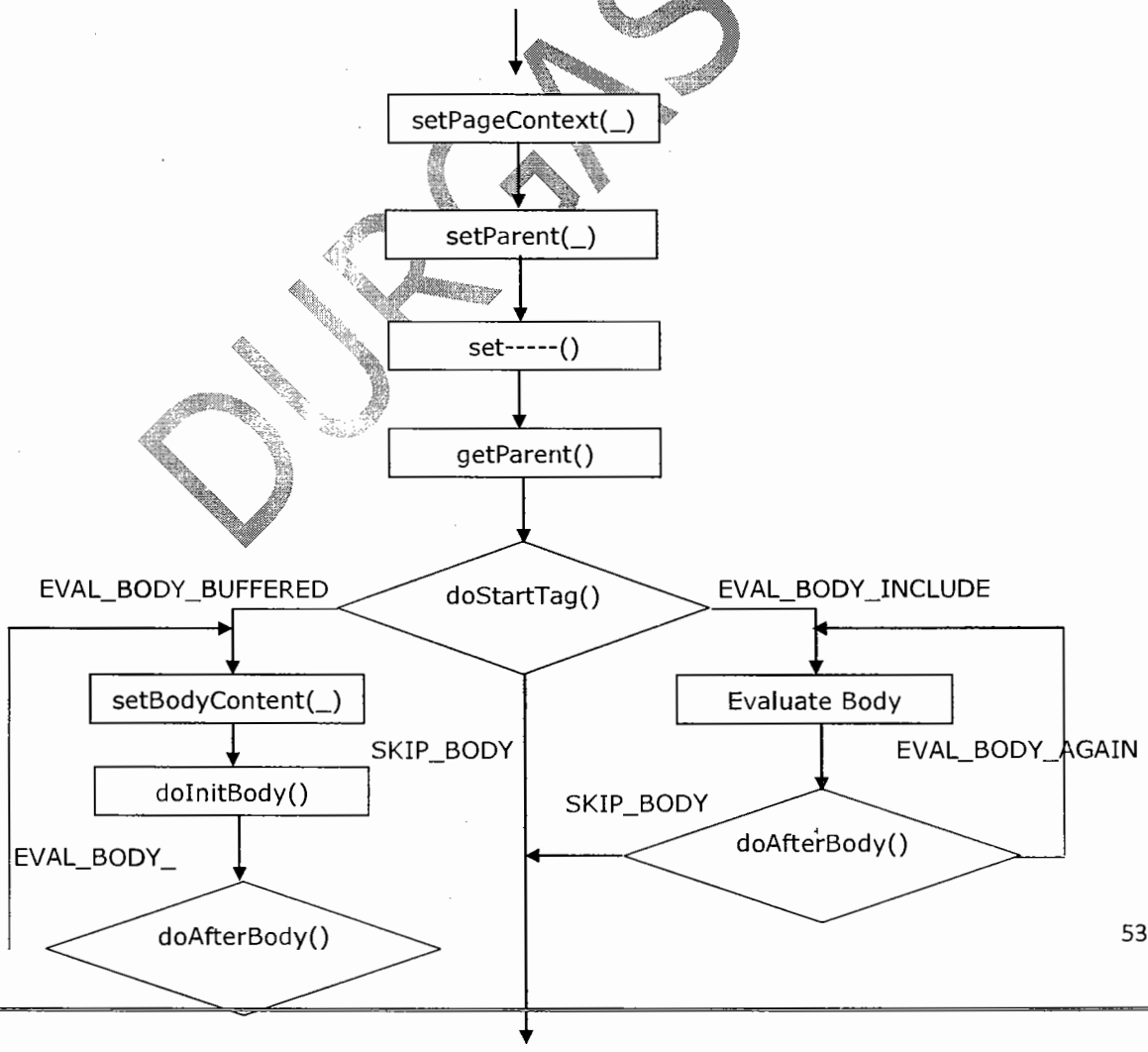
If we return SKIP_BODY constant from doStartTag() method then container will skip the custom tag body.

If we return EVAL_BODY_BUFFERED constant from doStartTag() methodthen container will store custom tag body in a buffer then access setBodyContent(_) method.

To access setBodyContent(_) method container will prepare BodyContent object with the buffer.

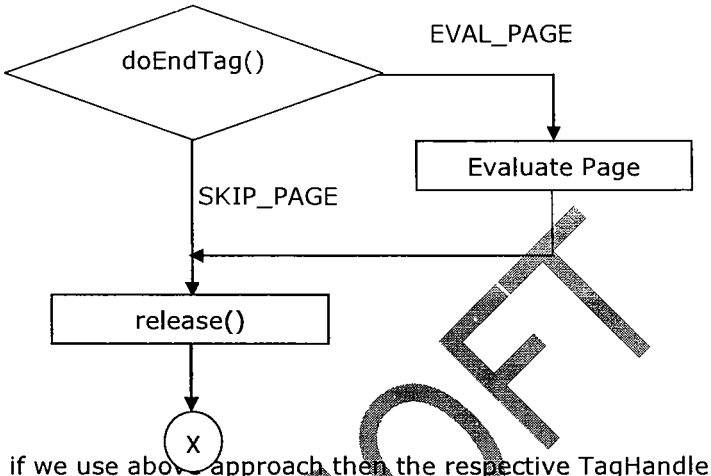
After executing setBodyContent(_) method container will access doInitBody() method in order to prepare BodyContent object for allow modifications.

Life Cycle of BodyTag interface:



53

INCLUDE SKIP_BODY



To prepare custom tags if we use above approach then the respective TagHandler class must implement all the methods declared in BodyTag interface irrespective of the application requirement.

This approach may increase burden to the developers and unnecessary methods in the custom tag application.

To overcome this problem we will use an alternative provided by Jsp technology i.e. javax.servlet.jsp.tagext.BodyTagSupport class.

BodyTagSupport class is a concrete class, a direct implementation class to BodyTag interface and it has provided the default implementation for all the methods declared in BodyTag interface.

If we want to prepare custom tags with BodyTagSupport class then the respective TagHandler class must extend BodyTagSupport and overrides the only required methods.

```
public classBodyTagSupport implements BodyTag {
   public static final int EVAL_BODY_INCLUDE;
   public static final int SKIP_BODY;
   public static final int EVAL_PAGE;
   public static final int SKIP_PAGE;
   public static final int EVAL_BODY_AGAIN;
   public static final int EVAL_BODY_BUFFERED;
   public PageContext pageContext;
   public Tag t;
   public BodyContent bodyContent;
   public void setPageContext(PageContext pageContext) {
      this.pageContext=pageContext;
   }
   public void setParent(Tag t) {
      this.t=t;
   }
   public Tag getParent() {
```

```
        return t;
    }
    public int doStartTag()throws JspException {
        return EVAL_BODY_BUFFERED;
    }
    public void setBodyContent(BodyContent bodyContent) {
        this.bodyContent=bodyContent;
    }
    public void doInitBody()throws JspException
    {  }
    public int doAfterBody()throws JspException {
        return SKIP_BODY;
    }
    public int doEndTag()throws JspException {
        return EVAL_PAGE;
    }
    public void release()
    {  }
}

  public class MyHandler extendsBodyTagSupport
{ ----- }
```

In case of body tags, custom tag body will be available in BodyContent object, to get custom tag body from BodyContent object we have to use the following method.

```
public String getString()
```

To send modified data to the response object we have to get JspWriter object from BodyContent, for this we have to use the following method.

```
public JspWriter getEnclosingWriter()
```

----------Application10----------

custapp5:

reverse.jsp:

```
<%@taglib uri="/WEB-INF/reverse.tld" prefix="mytags"%>
<mytags:reverse>
        Durga Software Solutions
</mytags:reverse>
```

reverse.tld:

```
<taglib>
        <jsp-version>2.1</jsp-version>
        <tlib-version>1.0</tlib-version>
        <tag>
                <name>reverse</name>
                <tag-class>com.dss.Reverse</tag-class>
                <body-content>jsp</body-content>
```

```
    </tag>
</taglib>
```

Reverse.java:

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class Reverse extends BodyTagSupport
{
    public int doEndTag() throws JspException
    {
        try
        {
            String data=bodyContent.getString();
            StringBuffer sb=new StringBuffer(data);
            sb.reverse();
            JspWriter out=bodyContent.getEnclosingWriter();
            out.println("<html>");
            out.println("<body bgcolor='lightyellow'>");
            out.println("<center><b><font size='7' color='red'>");
            out.println("<br><br>");
            out.println(sb);
            out.println("</font></b></center></body><html>");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        return EVAL_PAGE;
    }
}
```

----------Application11----------

custapp6:

editor.html:

```
<html>

<body bgcolor="lightgreen">
    <b><font size="6" color="red">
    <form action="./result.jsp">
    <pre>
        Enter SQL Query
        <textarea name="query" rows="5" cols="50"></textarea>
        <input type="submit" value="GetResult"/>
    </pre>
    </form></font></b></body>
```

```
</html>
```

result.jsp:

```
<%@taglib uri="/WEB-INF/result.tld" prefix="dbtags"%>
<jsp:declaration>
        String query;
</jsp:declaration>
<jsp:scriptlet>
        query=request.getParameter("query");
</jsp:scriptlet>
<dbtags:query>
        <jsp:expression>query</jsp:expression>
</dbtags:query>
```

result.tld:

```
<taglib>
        <jsp-version>2.1</jsp-version>
        <tlib-version>1.0</tlib-version>
        <tag>
                <name>query</name>
                <tag-class>com.dss.Result</tag-class>
                <body-content>jsp</body-content>
        </tag>
</taglib>
```

Result.java:

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.sql.*;
public class Result extends BodyTagSupport
{
        Connection con;
        Statement st;
        ResultSet rs;
        public Result()
        {
                try
                {
                        Class.forName("oracle.jdbc.driver.OracleDriver");

        con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","
durga");
                        st=con.createStatement();
                }
                catch (Exception e)
                {
                        e.printStackTrace();
                }
        }
```

```
public int doEndTag() throws JspException
{
	try
	{
		JspWriter out=pageContext.getOut();
		String query=bodyContent.getString();
		boolean b=st.execute(query);
		if (b==true)
		{
			rs=st.getResultSet();
			ResultSetMetaData rsmd=rs.getMetaData();
			out.println("<html>");
			out.println("<body bgcolor='lightblue'>");
			out.println("<center><br><br>");
			out.println("<table border='1' bgcolor='lightyellow'>");
			int count=rsmd.getColumnCount();
			out.println("<tr>");
			for (int i=1;i<=count;i++)
			{
				out.println("<td><center><b><font size='6'
color='red'>"+rsmd.getColumnName(i)+"</font></b></center></td>");
			}
			out.println("</tr>");
			while (rs.next())
			{
				out.println("<tr>");
				for (int i=1;i<=count;i++)
				{

out.println("<td><h1>"+rs.getString(i)+"</h1></td>");
				}
				out.println("</tr>");
			}
			out.println("</table></center></body></html>");
		}
		else
		{
			int rowCount=st.getUpdateCount();
			out.println("<html>");
			out.println("<body bgcolor='lightyellow'>");
			out.println("<center><b><font size='7' color='red'>");
			out.println("<br><br>");
			out.println("Record Updated : "+rowCount);

			out.println("</font></b></center></body></html>");
		}
	}
	catch (Exception e)
	{
		e.printStackTrace();
	}
	return EVAL_PAGE;
}
```

```
}
```

2. Simple Tags:

Q: What are the differences between Classic tags and Simple tags?

Ans: 1. Classic tags are more API independent, but Simple tags are less API independent.

2. If we want to design custom tags by using classic tag library then we have to remember 3 types of life cycles.

If we want to design custom tags by using simple tag library then we have to remember only 1 type of life cycle.

3. In case of classic tag library, all the TagHandler class objects are cacheable objects, but in case of simple tag library, all the TagHandler class objects are non-cacheable objects.

4. In case of classic tags, all the custom tags are not body tags by default, but in case of simple tags, all the custom tags are having body tags capacity by default.

If we want to design custom tags by using simple tag library then the respective TagHandler class must implement SimpleTag interface either directly or indirectly.

```
public interface SimpleTag extends JspTag {
   public void setJspContext(JspContext jspContext);
   public void setParent(JspTag t);
   public JspTag getParent();
   public void setJspBody(JspFragment jspFragment);
   public void doTag()throws JspException, IOException;
}
public class MyHandler implements SimpleTag
{ ------- }
```

Where jspContext is an implicit object available in simple tag library, it is same as pageContext, it can be used to make available all the Jsp implicit objects.
Where jspFragment is like bodyContent in classic tag library, it will accommodate custom tag body directly.

Where setJspContext(_) method can be used to inject JspContext object into the present web application.

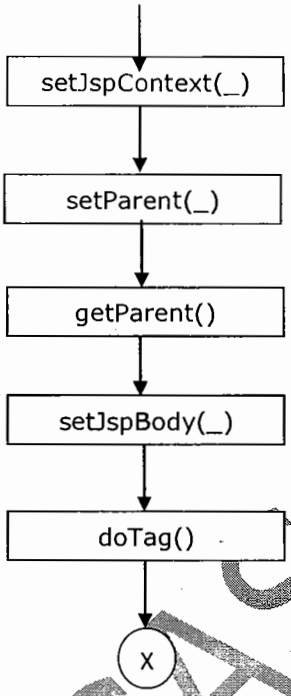
Where setParent(_) method can be used to inject parent tags TagHandler class object reference into the present TagHandler class.

Where getParent() method can be used to get parent tags TagHandler class object.

Where setJspBody(_) method is almost all equal to setBodyContent(_) method in order to accommodate custom tag body.

Where doTag() method is equivalent to doStartTag() method and doEndTag() method in order to perform an action.

Life Cycle of SimpleTag interface:



To design custom tags if we use approach then the respective TgaHandler class must implement SimpleTag interface i.e. it must implement all the methods which are declared in SimpleTag interface.

This approach will increase burden to the developers and unnecessary methods in the custom tags.

To overcome the above problem Jsp technology has provided an alternative in the form of javax.servlet.jsp.tagext.SimpleTagSupport class.

SimpleTagSupport is a concrete class provided by Jsp technology as an implementation class to SimpleTag interface with default implementation.

If we want to design custom tags by using SimpleTagSupport class then the respective TagHandler class must be extended from SimpleTagSupport class and where we have to override the required methods.

```
public interface SimpleTag extends JspTag {
    private JspContext jspContext;
    private JspFragment jspFragment;
    private JspTag jspTag;
    public void setJspContext(JspContext jspContext) {
       this.jspContext=jspContext;
    }
    public void setParent(JspTag t) {
       this.jspTag=jspTag;
```

```
    }
    public void setJspBody(JspFragment jspFragment) {
       this.jspFragment=jspFragment;
    }
    public JspTag getParent() {
       return jspTag;
    }
    public JspFragment getJspBody() {
       return jspFragment;
    }
    public JspContext getJspContext() {
       return jspContext;
    }
    public void doTag()throws JspException, IOException
    {  }
}
public class MyHandler extends SimpleTagSupport
{ ------- }
```

----------Application12----------

custapp7:

hello.jsp:

```
<%@taglib uri="/WEB-INF/hello.tld" prefix="mytags"%>
<mytags:hello/>
```

hello.tld:

```
<taglib>
        <jsp-version>2.1</jsp-version>
        <tlib-version>1.0</tlib-version>
        <tag>
                <name>hello</name>
                <tag-class>com.dss.Hello</tag-class>
                <body-content>empty</body-content>
        </tag>
</taglib>
```

Hello.java:

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
public class Hello extends SimpleTagSupport
{
        public void doTag() throws JspException,IOException
        {
                getJspContext().getOut().println("<h1><center>Hello SimpleTag
Application</center></h1>");
        }
}
```

Jsp Standard Tag Library(JSTL):

In Jsp technology, by using scripting elements we are able to provide Java code inside the Jsp pages.

To preserve Jsp principles we have to eliminate scripting elements, for this we have to use Jsp Actions.

In case of Jsp Actions, we will use standard actions as an alternative to scripting elements, but which are limited in number and having bounded functionality so that standard actions are not specified the required application format.

Still it is required to provide Java code inside the Jsp pages.

In the above context, to eliminate Java code completely from Jsp pages we have to use custom actions.

In case of custom actions, to implement simple Java syntaxes like if condition, for loop and so on we have to provide a lot of Java code internally.

To overcome the above problem Jsp technology has provided a separate tag library with simple Java syntaxes implementation and frequently used operations.

JSTL is an abstraction provided by Sun Microsystems, but where implementations are provided by all the server vendors.

With the above convention Apache Tomcat has provided JSTL implementation in the form of the jar files as standard.jar, jstl.jar.

Apache Tomcat has provided the above 2 jar files in the following location.

C:\Tomcat7.0\webapps\examples\WEB_INF\lib

If we want to get JSTL support in our Jsp pages then we have to keep the above 2 jar files in our web application lib folder.

JSTL has provided the complete tag library in the form of the following 5 types of tags.

1. Core Tags
2. XML Tags
3. Internationalization or I18N Tags (Formatted tags)
4. SQL Tags
5. Functions tags

To get a particular tag library support into the present Jsp page we have to use the following standard URL's to the attribute in the respective taglib directive.

http://java.sun.com/jstl/core

http://java.sun.com/jstl/xml

http://java.sun.com/jstl/fmt

http://java.sun.com/jstl/sql

http://java.sun.com/jsp/jstl/functions

1. Core Tags:

In JSTL, core tag library was divided into the following 4 types.

1. General Purpose Tags
    1. <c:set----->
    2. <c:remove----->
    3. <c:catch----->
    4. <c:out----->
2. Conditional Tags
    1. <c:if----->
    2. <c:choose----->
    3. <c:when----->
    4. <c:otherwise----->
3. Iterate Tags
    1. <c:forEach----->
    2. <c:forTokens----->
4. Url-Based Tags
    1. <c:import----->
    2. <c:url----->
    3. <c:redirect----->

1. General Purpose Tags:

1. <c:set----->:

This tag can be used to declare a single name value pair onto the specified scope.

Syntax:<c:set var="--" value="--" scope="--"/>

Where var attribute will take a variable i.e. key in key-value pair.

Where value attribute will take a particular value i.e. a value in key-value pair.

Where scope attribute will take a particular scope to include the specified key-value pair.

2. <c:out----->:

This tag can be used to display a particular value on the client browser.

Syntax:<c:out value="--"/>

Where value attribute will take a static data or an expression.

To present an expression with value attribute we have to use the following format.

Syntax:${expression}

Ex:<c:out value"${a}"/>

If the container encounters above tag then container will search for 'a' attribute in the page scope, request scope, session scope and application scope.

----------Application13----------

jstlapp1:

core.jsp:

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@page isELIgnored="true"%>
<html>
        <body>
                <center><b><font size="7">
                <c:set var="a" value="AAA" scope="request"/>
                <br>
                <c:out value="core tag library"/>
                <br><br>
                <c:out value="${a}"/>
                </font></b></center>
        <body>
<html>
```

3. <c:remove----->:

This tag can be used to remove an attribute from the specified scope.

Syntax:<c:remove var="--" scope="--"/>

Where scope attribute is optional, if we have not specified scope attribute then container will search for the respective attribute in the page scope, request scope, session scope and application scope.

----------Application14----------

core1.jsp:

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
		<c:set var="a" value="AAA" scope="request"/>
		<br>
		a-----><c:out value="${a}"/>
		<br><br>
		<c:remove var="a" scope="request"/>
		a-----><c:out value="${a}"/>
		</font></b></center>
	<body>
<html>
```

4. <c:catch----->:

This tag can be used to catch an Exception raised in its body.

Syntax:<c:catch var="--">

---------- </c:catch>

Where var attribute will take a variable to hold the generated Exception object reference.

----------Application15----------

core2.jsp:

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
		<c:catch var="e">
			<jsp:scriptlet>
				java.util.Date d=null;
				out.println(d.toString());
			</jsp:scriptlet>
		</c:catch>
		<c:out value="${e}"/>
		</font></b></center>
	<body>
<html>
```

2. Conditional Tags:

1. <c:if----->:

This tag can be used to implement if conditional statement.

Syntax:<c:if test="--"/>

Where test attribute is a boolean attribute, it may take either true or false values.

----------Application16----------

core3.jsp:

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
			<c:set var="a" value="10"/>
			<c:set var="b" value="20"/>
			<c:if test="${a<b}">
				condition is true
			</c:if>
			<br>
			out of if
		</font></b></center>
	<body>
<html>
```

2. <c:choose----->, <c:when-----> and <c:otherwise----->:

These tags can be used to implement switch programming construct.

Syntax:

```
<c:choose>
<c:when test="--">
		----------
</c:when>
		----------
<c:otherwise>
		----------
</c:otherwise>
</c:choose>
```

----------Application17----------

core4.jsp:

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

```
        <body>
                <center><b><font size="7">
                        <c:set var="a" value="10"/>
                        <c:choose>
                                <c:when test="${a==10}">
                                        TEN
                                </c:when>
                                <c:when test="${a==15}">
                                        FIFTEEN
                                </c:when>
                                <c:when test="${a==20}">
                                        TWENTY
                                </c:when>
                                <c:otherwise>
                                        Number is not in 10,15 and 20
                                </c:otherwise>
                        </c:choose>
                </font></b></center>
        <body>
<html>
```

3. Iterator Tags:

1. <c:forEach----->:

This tag can be used to implement for loop to provide iterations on its body and it can be used to perform iterations over an array of elements or Collection of elements.

Syntax 1:<c:forEach var="--" begin="--" end="--" step="--">

```
            ------------
</c:forEach>
```

Where var attribute will take a variable to hold up the loop index value at each and every iteration.

Where begin and end attribute will take start index value and end index value.

Syntax 2:<c:forEach var="--" items="--">

```
            ------------
</c:forEach>
```

Where var attribute will take a variable to hold up an element from the respective Collection at each and every iteration.

Where items attribute will take the reference of an array or Collection from either of the scopes page, request, session and application.

----------Application18----------

core5.jsp:

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
			<c:forEach var="a" begin="0" end="10" step="2">
				<c:out value="${a}"/><br>
			</c:forEach>
			<%
				String[] s={"A","B","C"};
				request.setAttribute("s",s);
			%>
			<br>
			<c:forEach var="x" items="${s}">
				<c:out value="${x}"/><br>
			</c:forEach>
		</font></b></center>
	<body>
<html>
```

2. <c:forTokens----->:

This tag can be used to perform String Tokenization on a particular String.

Syntax :<c:forTokens var="--" items="--" delims="--">

</c:forTokens>

Where var attribute will take a variable to hold up token at each and every iteration.

Where items attribute will take a String to tokenize.

Where delims attribute will take a delimiter to perform tokenization.

----------Application19----------

core6.jsp:

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

```
            <center><b><font size="7">
                <c:forTokens var="token" items="Durga Software Solutions" delims="
">
                    <c:out value="${token}"/><br>
                </c:forTokens>
            </font></b></center>
        <body>
<html>
```

4. Url-Based Tags:

1. <c:import----->:

This tag can be used to import the content of the specified target resource into the present Jsp page.

----------Application20----------

second.jsp:

```
<center><h1>This is second.jsp</h1></center>
```

core7.jsp:

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

<%@page isELIgnored="true"%>
<html>
        <body>
            <center><b><font size="7">
                Start
                <br>
                <c:import url="second.jsp"/>
                <br>
                End
            </font></b></center>
        <body>
<html>
```

2. <c:url----->:

This tag can be used to represent the specified url.

Syntax :<c:url value="--"/>

----------Application21----------

core8.jsp:

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
			<a href='<c:url value="http://localhost:2020/loginapp"/>'>Login
Application</a>
		</font></b></center>
	<body>
<html>
```

2. <c:redirect----->:

This tag can be used to implement Send Redirect Mechanism from a particular Jsp page.

Syntax:<c:redirect url="--"/>

----------Application22----------

core9.jsp:

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

<html>
	<body>
		<center><b><font size="7">
			<c:redirect url="http://localhost:2020/registrationapp"/>
		</font></b></center>
	<body>
<html>
```


4. SQL Tags:

The main purpose of SQL tag library is to interact with the database in order to perform the basic database operations.

JSTL has provided the following set of tags as part of SQL tag library.

1. <sql:setDataSource----->

2. <sql:update----->
3. <sql:query----->
4. <sql:transaction----->
5. <sql:param----->
6. <sql:dateParam----->

1. <sql:setDataSource----->:

This tag can be used to prepare the Jdbc environment like Driver loading, establish the connection.

Syntax:<sql:setDataSource driver="--" url="--" user="--" password="--"/>

Where driver attribute will take the respective Driver class name.

Where url attribute will take the respective Driver url.

Where user and password attributes will take database user name and password.

2. <sql:update----->:

This tag can be used to execute all the updation group SQL queries like create, insert, update, delete, drop and alter.

Syntax 1:<sql:update var="--" sql="--"/>

Syntax 2:<sql:update var="--"> ------ query ------ </sql:update>

In the above <sql:update> tag we are able to provide the SQL queries in 2 ways.

1. Statement style SQL queries, which should not have positional parameters.
2. PreparedStatement style SQL queries, which should have positional parameters.

If we use PreparedStatement style SQL queries then we have to provide values to the positional parameters, for this we have to use the following SQL tags.

1. <sql:param----->:

This tag can be used to set a normal value to the positional parameter.

Syntax 1:<sql:param value="--"/>

Syntax 2:<sql:param> ------ value ------ </sql:param>

2. <sql:dateParam----->:

This tag can be used to set a value to parameter representing data.

Syntax 1:<sql:dateParam value="--"/>

Syntax 2:<sql:dateParam>value</sql:dateParam>

----------Application23----------

jstlapp2:

sql.jsp:

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
			<sql:setDataSource driver="oracle.jdbc.driver.OracleDriver"
url="jdbc:oracle:thin:@localhost:1521:xe" user="system" password="durga"/>
			<sql:update var="result" sql="create table emp1(eid number, ename
varchar2(10), esal number)"/>
			Row Count ..... <c:out value="${result}"/>
		</font></b></center>
	</body>
</html>
```

----------**Application24**----------

sql1.jsp:

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
			<sql:setDataSource driver="oracle.jdbc.driver.OracleDriver"
url="jdbc:oracle:thin:@localhost:1521:xe" user="system" password="durga"/>
			<sql:update var="result" sql="insert into emp1
values(101,'aaa',5000)"/>

			Row Count ..... <c:out value="${result}"/>
		</font></b></center>
	</body>
</html>
```


----------**Application25**----------

sql2.jsp:

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

```
<%@taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
			<sql:setDataSource driver="oracle.jdbc.driver.OracleDriver"
url="jdbc:oracle:thin:@localhost:1521:xe" user="system" password="durga"/>
			<sql:update var="result" sql="insert into emp1 values(?,?,?)">
				<sql:param value="103"/>
				<sql:param>ccc</sql:param>
				<sql:param value="7000"/>
			</sql:update>
			Row Count ..... <c:out value="${result}"/>
		</font></b></center>
	</body>
</html>
```

----------Application26----------

sql3.jsp:

```
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
			<sql:setDataSource driver="oracle.jdbc.driver.OracleDriver"
url="jdbc:oracle:thin:@localhost:1521:xe" user="system" password="durga"/>
			<sql:update var="result">
				update emp1 set esal=esal+? where esal>?
				<sql:param>500</sql:param>
				<sql:param>5000</sql:param>
			</sql:update>
			Row Count ..... <c:out value="${result}"/>


</font></b></center></body>
</html>
```


----------Application27----------

sql4.jsp:

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

```
<%@taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
			<sql:setDataSource driver="oracle.jdbc.driver.OracleDriver"
url="jdbc:oracle:thin:@localhost:1521:xe" user="system" password="durga"/>
			<sql:update var="result">
				delete emp where esal>1000
			</sql:update>
			Row Count ..... <c:out value="${result}"/>
		</font></b></center>
	</body>
</html>
```

3. <sql:query----->:

This tag can be used to execute selection group SQL queries in order to fetch the data from database table.

Syntax 1:<sql:query var="--" sql="--"/>

Syntax 2:<sql:query var="--"> ----- query ----- </sql:query>

If we execute selection group SQL queries by using <sql:query> tag then SQL tag library will prepare result object to hold up fetched data.

In SQL tag library, result object is a combination of ResultSet object and ResultSetMetaData object.

In result object, all the column names will be represented in the form a single dimensional array referred by columnNames predefined variable and column data (table body) will be represented in the form of 2-dimensionnal array referred by rowsByIndex predefined variable.

----------Application28----------

sql5.jsp:

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

<%@taglib uri="http://java.sun.com/jstl/sql" prefix="sql"%>
<%@page isELIgnored="true"%>
<html>
	<body>
		<center><b><font size="7">
			<sql:setDataSource driver="oracle.jdbc.driver.OracleDriver"
url="jdbc:oracle:thin:@localhost:1521:xe" user="system" password="durga"/>
			<sql:query var="result" sql="select * from emp"/>
			<table border="1" bgcolor="lightyellow">
			<tr>
```

```
                        <c:forEach var="columnName"
items="${result.columnNames}">
                                <td><center><b><font size="6" color="red">
                                <c:out value="${columnName}"/>
                                </font></b></center></td>
                        </c:forEach>
                </tr>
                <c:forEach var="row" items="${result.rowsByIndex}">
                        <tr>
                                <c:forEach var="column" items="${row}">
                                        <td><b><font size="5">
                                                <c:out value="${column}"/>
                                        </font></b></td>
                                </c:forEach>
                        </tr>
                </c:forEach>
                </table>
        </font></b></center>
    </body>
</html>
```

4. <sql:transaction----->:

This tag will represent a transaction, which includes collection of <sql:update> tags and <sql:query> tags.

3. I18N Tags(Formatted Tags):

1. <fmt:setLocale----->:

This tag can be used to represent a particular Locale.

Syntax:<fmt:setLocale value="--"/>

Where value attribute will take Locale parameters like en_US, it_IT and so on.

2. <fmt:formatNumber----->:

This tag can be used to represent a number w.r.t the specified Locale.

Syntax:<fmt:formatNumber var="--" value="--"/>

Where var attribute will take a variable to hold up the formatted number.

Where value attribute will take a number.

3. <fmt:formatDate----->:

This tag can be used to format present system date w.r.t. a particular Locale.

Syntax:<fmt:formatDate var="--" value="--"/>

Where var attribute will take a variable to hold up the formatted date.

Where value attribute will take the reference of Date object.

4. <fmt:setBundle----->:

This tag can be used to prepare ResourceBundle object on the basis of a particular properties file.

Syntax:<fmt:setBundle var="--" basename="--"/>

Where var attribute will take a variable to hold up ResourceBundle object reference.

Where basename attribute will take base name of the properties file.

5. <fmt:message----->:

This tag can be used to get the message from ResourceBundle object on the basis of provided key.

Syntax:<fmt:message var="--" key="--"/>

Where var attribute will take a variable to hold up message.

Where key attribute will take key of the message defined in the respective properties file.

----------Application29----------

jstlapp3:

abc_en_US.properties:

```
#abc_en_US.properties
#-----------------------------
welcome=Welcome to US user
```

abc_it_IT.properties:

```
#abc_it_IT.properties
#-----------------------------
welcome=Welcome toe Italiano usereo
```

fmt.jsp:

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

<%@taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"%>
<%@page isELIgnored="true"%>
<html>
        <body>
                <center><b><font size="7">
                        <fmt:setLocale value="it_IT"/>
                        <fmt:formatNumber var="num" value="123456.789"/>
                        <c:out value="${num}"/><br><br>
```

```
                    <jsp:useBean id="date" class="java.util.Date">
                        <fmt:formatDate var="fdate" value="${date}"/>
                        <c:out value="${fdate}"/>
                    </jsp:useBean>
                    <br><br>
                    <fmt:setBundle basename="abc"/>
                    <fmt:message var="msg" key="welcome"/>
                    <c:out value="${msg}"/>
                </font></b></center>
        </body>
</html>
```

JSTL Functions:

The main purpose of functions tag library is to perform all the String operations which are defined in the String class.

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

<%taglib uri="http://java.sun.com/jstl//jspfunctions" prefix="fn"%>

<c:set var="a" value="Durga Software Solutions"/>

${fn.length(a)}

${fn.concat(a, "Hyderabad")}

${fn.toLowerCase(a)}

${fn.toUpperCase (a)}

${fn.contains(a, "Software")}

${fn.startsWith(a, "Durga")}

${fn.endsWith(a, "Solutions")}

${fn.substring(a, 4, 20)}

Expression Language:

In JSP technology,scripting elements will allow java code inside the JSP pages,which is against to JSP principles.

To eliminate java code and scripting elements from JSP pages,we have to use standard actions,but which are in limited number with bounded functionality.

We will use custom actions,but it should require lot of java code internally inorder to implement simple programming constructs like if,switch,for and so on.

To overcome the above problems,we are able to use JSTL tag library,but still it is not sufficient to eliminate java code completely from JSP pages.

In the above context,to eliminate java code completely from JSP pages,we have to use Expression language syntax along with standard actions,custom actions and JSTL tag library.

Eg:

To get a particular request parameter from request object,we have to use java code before expression language.

```
<%
uname=request.getParameter("uname");
%>
```

In the above context,we are able to retrieve a particular request parameter without using java code, by using expression language syntax.

Ex:

```
${param.uname}
```

To eliminate java code completely from JSP pages,by using the following expression language elements.

1)EL operators

2)EL implicit objects.

3)EL functions.

1)EL operators:

To prepare expressions in expression language,we have to use the following operators.

-->General purpose operators:

.,(),{],[]

-->Arithmetic Operators:

+ - * / % and so on

-->Comparison Operators:

== or eq

!= or ne

< or lt

> or gt

<= or le

>= or ge

-->Ternary Operators:

expr1?expr2:expr3;

-->Logical Operators:

&& ||

2)EL implicit objects:

-->Scope related:

PageScope

requestScope

sessionScope

applicationScope

param

paramValues

initParam --->to get context parameters

cookie

header --->to get particular header value

headerValues

pageContext

In JSP technology,we will use JSP implicit objects in scripting elements to reduce java codeinside the scripting elements.

Similarly,Expression Language has provided the following list of implicit objects to eliminate Java code from JSP pages.

pageScope,requestScope,sessionScope,applicationScope,param,paramValues, initParam,cookie,header,headerValues,pageContext.

To access the attributes data from the page scope,request scope,session scope and application scope,we will use the above scope related implicit objects like pageScope,requestScope,sessionScope and applicationScope implicit objects respectively.

Ex:

```
<%
request.setAttribute("uname","Laddu");
%>
<html>
<body>
${requestScope.uname}
</body>
</html>
```

param:

This can be used to access a particular request parameter.

paramValues:

This implicit object can be used to access more than one parameter values associated with a particular request parameter.

Ex:

form.html:

```
<html>
<body>
<center><b><font size="6">
<form method="get" action="first.jsp">
<br><br>
Name <input type="text" name="uname"/><br><br>
Food items <select size="3" multiple="true" name="food">
<option value="Dosa">Dosa</option>
<option value=="Vada">Vada</option>
<option value="Idly">Idly</option>
</select><input type="submit" value="Display"/>
</form></font></b></center>
</body>
</html>
```

first.jsp:

```
<html>
<body>
<center><b><font size="7"><br><br>
User Name.....${param.uname}
<br><br>
Food Items...<br>
${paramValues.food[0]}<br>
${paramValues.food[1]}<br>
${paramValues.food[2]}<br>
</font></b></center>
</body>
</html>
```

initParam

This implicit object can be used to access context parameters from ServletContex object.

Ex:

Web.xml:

```
<web-app>
<context-param>
<param-name>a</param-name>
<param-name>AAA</param-name>
</context-param>
        <context-param>
<param-name>b/param-name>
```

```
<param-name>BBB</param-name>
</context-param>
</web-app>
```

first.jsp:

```
<html>
<body>
<center><b><font size="7">
a--------->${initParam.a}<br>
b--------->${initParam.b}
</font></b></center>
</body>
</html>
```

cookie:

This implicit object can be used to get session id cookie name and session id cookie value.

Ex:

```
<html>
<body>
${cookie.JSESSIONID.name}<br>
${cookie.JSESSIONID.value}
</body>
</html>
```

header:

This implicit object can be used to access a particular request header value

Ex:

${header.cookie}

headerValues:

This implicit object can be used to access more than one value associated with a single request header.

Ex:

${headerValues.accept[0]}

pageContext:

To get all the implicit objects and to get the values from either of the scopes like pageScope,requestScope,sessionScope,applicationScope we will use pageContext implicit object.

Ex:

${pageContext.servletContext.servletInfo}

Expression Language Functions:

The main purpose of language functions is to perform a particular action in web applications.

In web applications we will utilise expression language functions as an alternative to JSP custom actions.

To prepare functions in expression language,we have to use the following steps:

1)Define a function by taking Java class under classes floder:

In expression language functions,we have to define a java method as a static method with the required implementation.

Ex:

```
public class Hello
{
public static String sayHello(String name)
{
return "Good Morning..."+name;
}
}
```

2)Configure expression language function in TLD file:

```
<taglib>
------
------
<function>
<name>function_name</name>
<function_class>Class_name</function_class>
<function_signature>method_signature</function_signature>
</function>
</taglib>
```

3)Access the function from JSP page:

To access the function in JSP page,we have to use the following syntax:

```
${fn:function_name([param_list])}
```

Ex:

```
${fn:sayHello("Nag")}
```

Ex:

firstapp

|

|---------first.jsp

|

|---------WEB-INF

| |

| |----hello.tld

| |----classes

| | |---com.dss.Hello.class

| | |

|

Hello.class

```
package com.dss;
public class Hello
{
public static String sayHello(String name)
{
return name;
}
}
```

hello.tld:

```
<taglib>
<jsp-version>2.1</jsp-version>
<tlib-version>1.0</tlib-version>
<function>
<name>sayHello</name>
<function-class>com.dss.Hello</function-class>
<function-signature>java.lang.String sayHello(java.lang.String)</function-signature>
</function>
</taglib>
```

first.jsp:

```
<%taglib uri="/WEB-INF/hello.tld" prefix="fn"%>
<html>
<body>
<b><fint size="7"><br><br>
${fn:sayHello("durga")}
</font></b>
</body>
</html>
```

JSP INTERVIEW QUESTIONS:

1. What is JSP ? Describe its concept.

2 . Explain the benefits of JSP?

3. Is JSP technology extensible?

4 .Can we implement an interface in a JSP?

5 What are the advantages of JSP over Servlet?

6. Differences between Servlets and JSP?

7 . Explain the differences between ASP and JSP?

8 . Can I stop JSP execution while in the midst of processing a request?

9. How to Protect JSPs from direct access ?

10. Explain JSP API ?

11. What are the lifecycle phases of a JSP?

12. Explain the life-cycle mehtods in JSP?

13. Difference between _jspService() and other life cycle methods.

14 What is the jspInit() method?

15. What is the _jspService() method?

16. What is the jspDestroy() method?

17. What JSP lifecycle methods can I override?

18. How can I override the jspInit() and jspDestroy() methods within a JSP page?

19 . Explain about translation and execution of Java Server pages?

20 . Why is _jspService() method starting with an '_' while other life cycle methods do not?

21. How to pre-compile JSP?

22. The benefits of pre-compiling a JSP page?

23.How many JSP scripting elements and explain them?

24. What is a Scriptlet?

25. What is a JSP declarative?

26. How can I declare methods within my JSP page?

27. What is the difference b/w variable declared inside a declaration and variable declared in scriplet ?

28.What are the three kinds of comments in JSP and what's the difference between them?

29. What is output comment?

30. What is a Hidden Comment?

31. How is scripting disabled?

32. What are the JSP implicit objects?

33. How does JSP handle run-time exceptions?

34. How can I implement a thread-safe JSP page? What are the advantages and Disadvantages of using it?

35. What is the difference between ServletContext and PageContext?

36 . Is there a way to reference the "this" variable within a JSP page?

37 . Can you make use of a ServletOutputStream object from within a JSP page?

38 .What is the page directive is used to prevent a JSP page from automatically creating a session?

39. What's a better approach for enabling thread-safe servlets and JSPs? SingleThreadModel Interface or Synchronization?

40. What are various attributes Of Page Directive ?

41 . Explain about autoflush?

42. How do you restrict page errors display in the JSP page?

43. What are the different scopes available fos JSPs ?

44. when do use application scope?

45. What are the different scope valiues for the ?

46. How do I use a scriptlet to initialize a newly instantiated bean?

47 . Can a JSP page instantiate a serialized bean?

48.How do we include static files within a jsp page ?

49.In JSPs how many ways are possible to perform inclusion?

50.In which situation we can use static include and dynamic include in JSPs ?

51.Differences between static include directive and include action ?