JSP
===

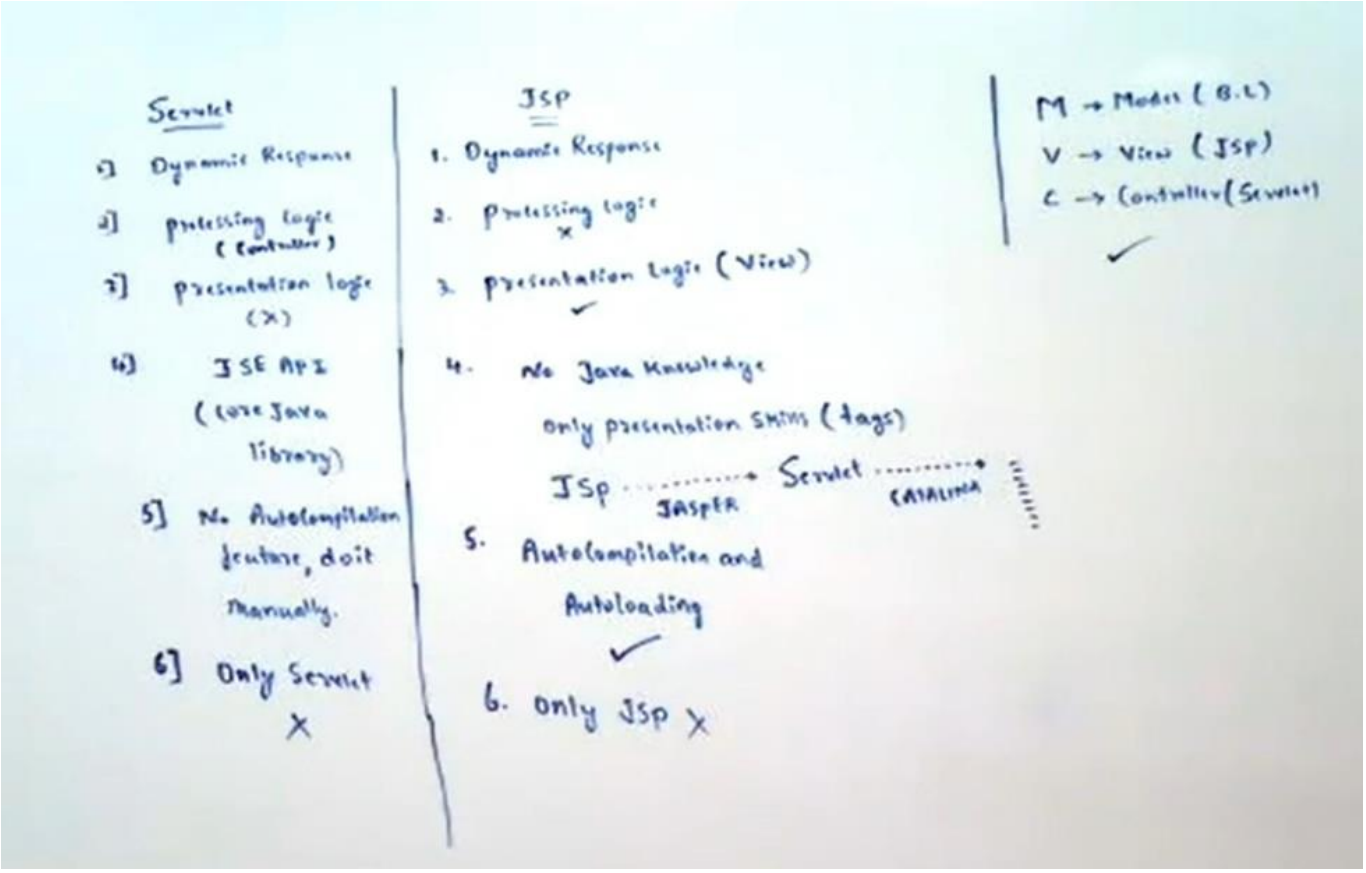
1. Introduction
2. JSP Lifecycle
3. JSP Directives
    a. JSP Page Directive
    b. JSP Include Directive
    c. JSP Taglib Directive
4. JSP Scripting Elements
    a. Declarations
    b. Scriptlets
    c. Expression
5. JSP Scopes
    a. Page Scope
    b. Request Scope
    c. Application Scope
    d. Session Scope
6. Implicit Objects(9)
7. JSP Actions
    a. Standard Actions
    b. Custom Actions
8. Action Tags
    a. <jsp:useBean>
    b. <jsp:setProperty>
    c. <jsp:getProperty>
    d. <jsp:include>
    e. <jsp:forward>
    f. <jsp:param>
    g. <jsp:plugin>
    h. <jsp:params>
    i. <jsp:declaration>
    j. <jsp:scriptlet>
    h. <jsp:expression>
9. Custom Tags
    a. Classic Tags
    b. Simple Tags

JSTL
====

1. Introduction
2. Core Tags
3. XML Tags
4. SQL Tags
5. Formatted Tags
6. Functions Tags

Expression Language
==========

1. Introduction
2. EL implicit Objects
3. EL Functions



JSP(JAVA SERVER PAGES)

==========

JAVA

a. Standalone Application(no client-server architecture)
    a. CUI Application
    b. GUI Application
b. Enterprise Application(client-server architecture)
    a. Web Application
        eg: Facebook,ABC website(www.abcfortech.com),.....
    b. Distrubuted Application(remote services)
        eg: Flipkart,Myntra,BookyMyShow,....

=> 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

====

=> 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.

Servlet

=======

=> 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.

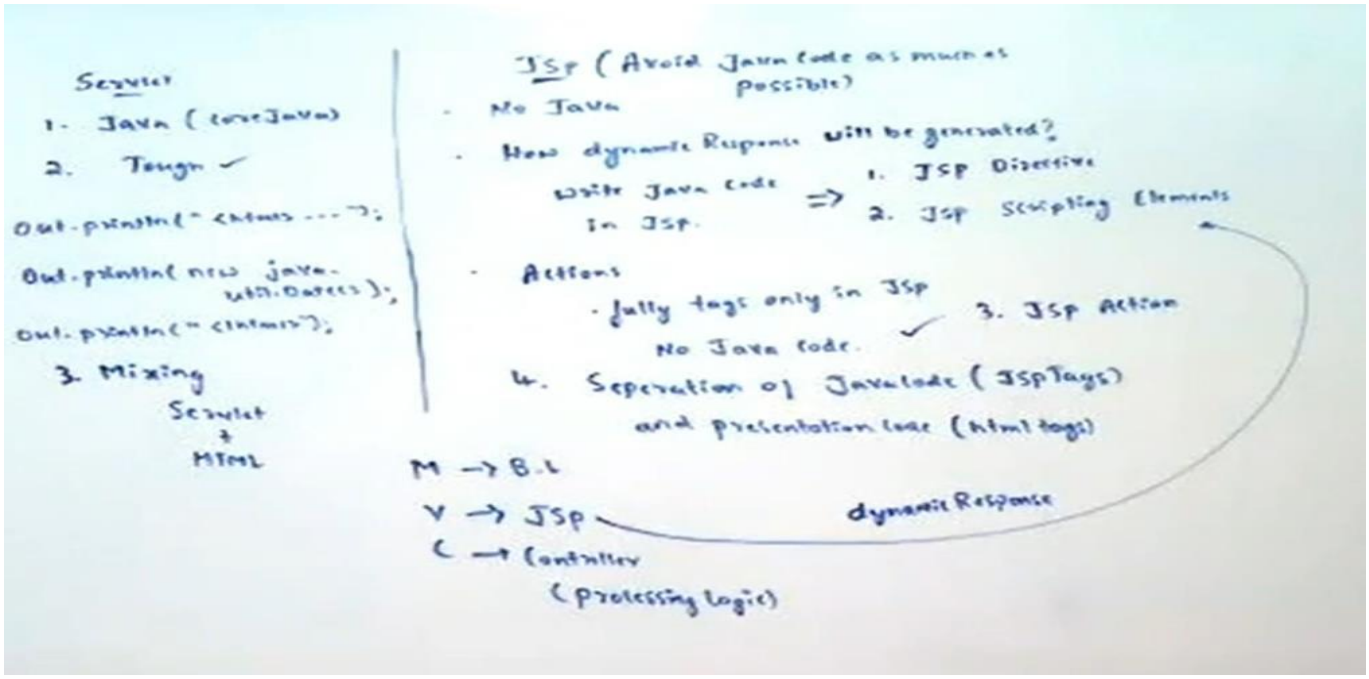
=> Servlet is a Thread based technology, if we deploy it at server then container will create a separate thread instead of the process for every request from the client.

=> 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

===

=> 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.

JSP --jasper---> **Servlet** --catalina----> processed to generate dynamic response.

=> 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

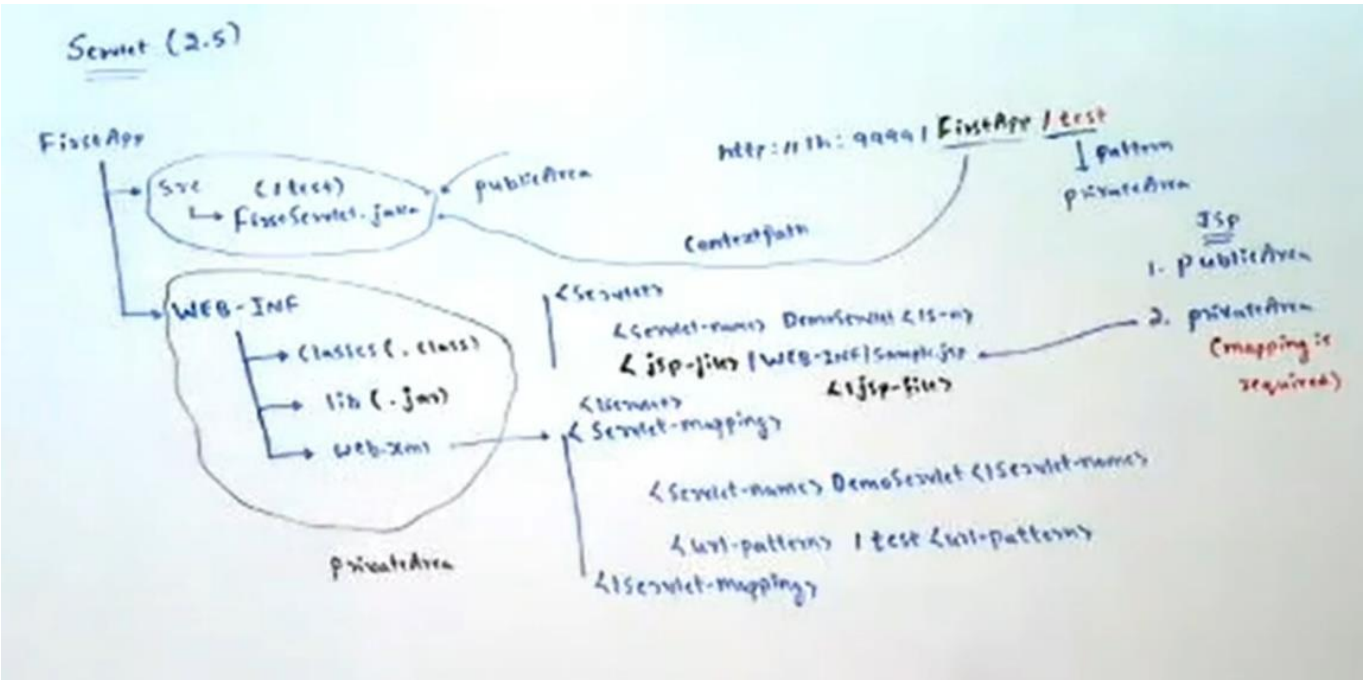
==========

=> 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 then we 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>
```

eg:

```
<web-app>
      <servlet>
            <servlet-name>WelcomeServlet</servlet-name>
            <Jsp-file>/WEB-INF/hellopage.jsp</Jsp-file >
      </servlet>
      <servlet-mapping>
            <servlet-name>HelloServlet</servlet-name>
            <url-pattern>/hel</url-pattern>
      </servlet-mapping>
</web-app>
```

1.hel--------> /WEB-INF/hellopage.jsp

eg:

```
<web-app>
      <servlet>
            <servlet-name>WelcomeServlet</servlet-name>
            <Jsp-file>/WEB-INF/classes/wishpage.jsp</Jsp-file >
      </servlet>
      <servlet-mapping>
            <servlet-name>WishServlet</servlet-name>
            <url-pattern>/wish</url-pattern>
      </servlet-mapping>
</web-app>
```

1.wish--------> /WEB-INF/classes/wishpage.jsp

Example-1

1. welcomepage.jsp -> Application folder
2. hellopage.jsp -> WEB-INF
3. wishpage.jsp -> WEB-INF/classes

refer: jspapp01

JSP API defines the following 2 interfaces.
1. JspPage(I)
2. HttpJspPage(I)

```
C:\Users\ABC>javap javax.servlet.jsp.HttpJspPage
Compiled from "HttpJspPage.java"
public interface javax.servlet.jsp.HttpJspPage extends javax.servlet.jsp.JspPage {
  public abstract void _jspService(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse) thro
;
}

C:\Users\ABC>javap javax.servlet.jsp.JspPage
Compiled from "JspPage.java"
public interface javax.servlet.jsp.JspPage extends javax.servlet.Servlet {
  public abstract void jspInit();
  public abstract void jspDestroy();
}

C:\Users\ABC>
```

1) JspPage(I):

This interface present in javax.servlet.jsp package and defines the following 2 life cycle methods

1) jspInit()
2) jspDestroy()

1) jspInit():

This method will be executed only once at the time of first request to perform initialization activities.

=> Web container always calls init(ServletConfig) of HttpJspBase class which internally calls jspInit() method.

```
public abstract class HttpJspBase ..{
        public final void init(ServletConfig config).. {
                jspInit();
        }
}
```

=> Based on our requirement we can override jspInit() method in the jsp to define our own initialization activities...

test.jsp:

```
<%!
        public void jspInit() {
                System.out.println("jsp initialization activities");
        }
 %>
        <h1>The Server Time is:<%= new java.util.Date() %></h1>
```

Note:
We cannot place init(ServletConfig config) in the JSP b'z it is declared as final in HttpJspBase class.

2) jspDestroy():
=> This method will be executed only once to perform cleanup activities just before taking jsp from out of service.

=> We can override jspDestroy() method in our jsp to define our own cleanup activities.

=> Web container always calls destroy() method available in HttpJspBase class which internally calls our jspDestroy() method.

```
C:\Users\ABC>javap org.apache.jasper.runtime.HttpJspBase
Compiled from "HttpJspBase.java"
public abstract class org.apache.jasper.runtime.HttpJspBase extends javax.servlet.http.HttpServlet implements javax.servlet.js
  protected org.apache.jasper.runtime.HttpJspBase();
  public final void init(javax.servlet.ServletConfig) throws javax.servlet.ServletException;
  public java.lang.String getServletInfo();
  public final void destroy();
  public final void service(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse) throws javax.servle
  public void jspInit();
  public void _jspInit();
  public void jspDestroy();
  protected void _jspDestroy();
  public abstract void _jspService(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse) throws javax
;
}

C:\Users\ABC>
```

```
public abstract class HttpJspBase .. {
        public final void destroy() {
                jspDestroy();
        }
}
```

test.jsp:

```
<%!
        public void jspDestroy(){
                System.out.println("jsp cleanup activities");
        }
%>
  <h1>The Server Time is:<%= new java.util.Date() %></h1>
```

Note:
We cannot write destroy() method directly in the JSP b'z it is declared as final in HttpJspBase class.

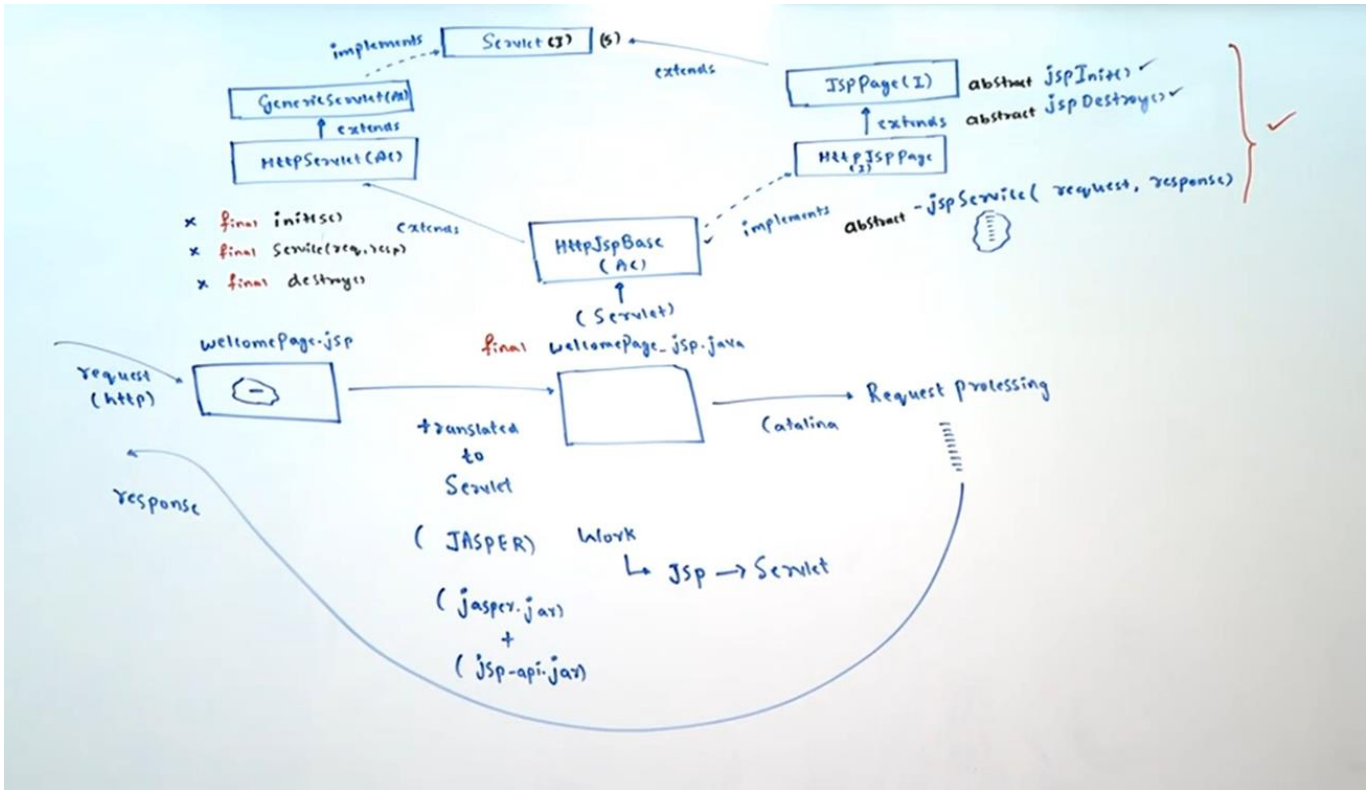
II) HttpJspPage(I):

=> It is the child interface of JspPage.
=> It defines only one method _jspService() method.
=> This method will be executed seperately for every request.
=> Web container always calles service() method present in HttpJspBase class which internally calls our service()method.

```
        public abstract class HttpJspBase extends.. {
                public final void service(HSR request, HSR response) throws SE, IE {
                        _jspService(request, response);
                 }
        }
```

=> At the time of translation, JSP Engine will place this method in the generated servlet class.

refer: Jspapp02



Q1.In the JSP is it possible to write _jspService() method explicitly?

No b'z this method already generated by JSP Engine automatically.
If we write explicitly then generated servlet will contain two _jspService() methods,which causes compile time error.

Q2. What is the significance of _ symbol in _jspService() method?

It indicates that this method will be generated automatically by JSP Engine and we cannot write explicitly.

Q3. Is it possible to write service() method explicitly in the JSP?

No b'z this method declared as final in HttpJspBase class.

Note:
JspPage
1. jspInit()
2. jspDestroy()

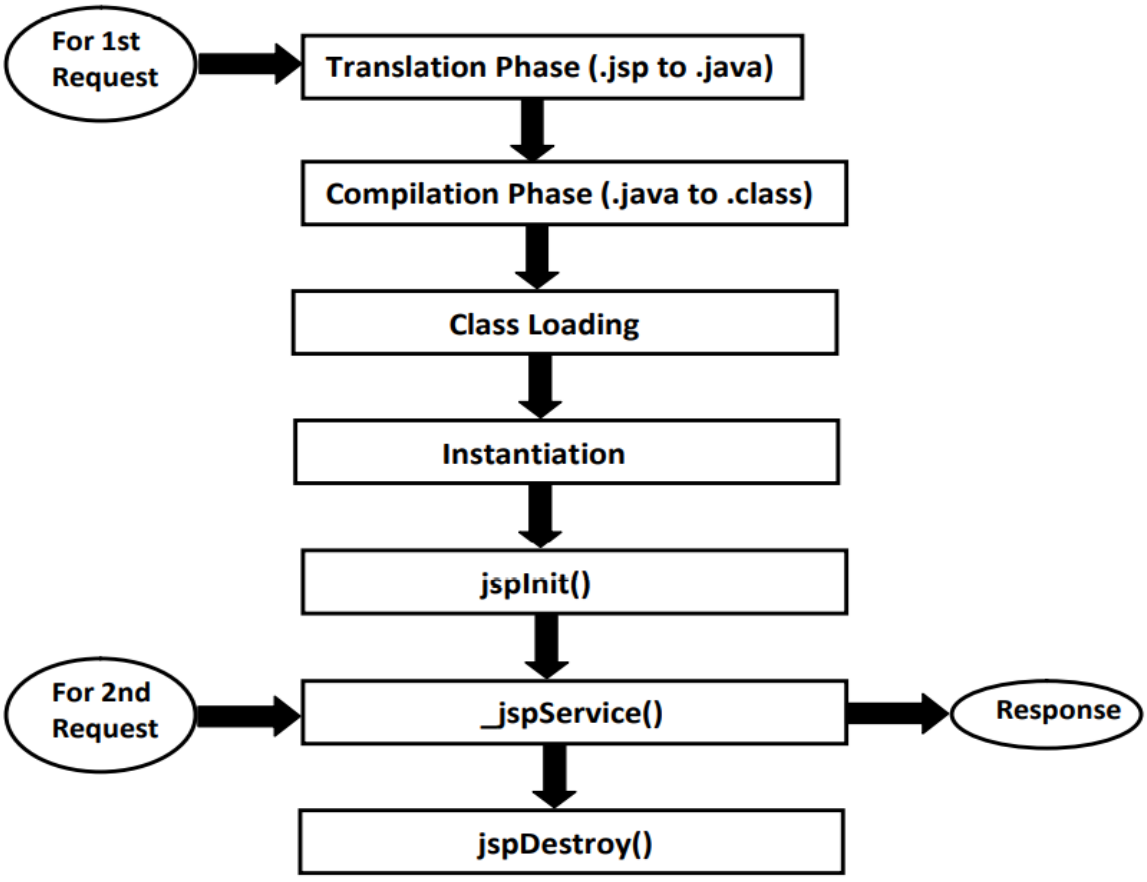
HttpJspPage
1. _jspService()

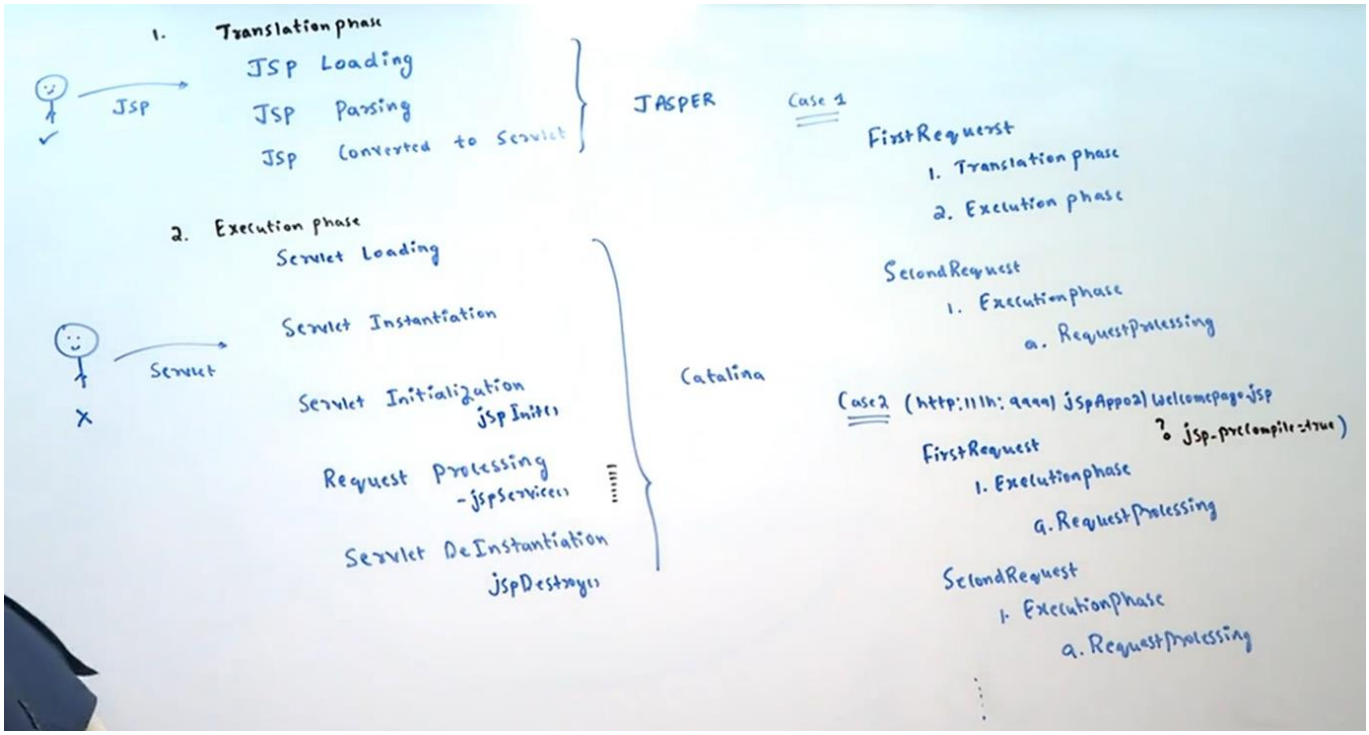
Q.In our JSP, which of the following methods we can write explicitly?

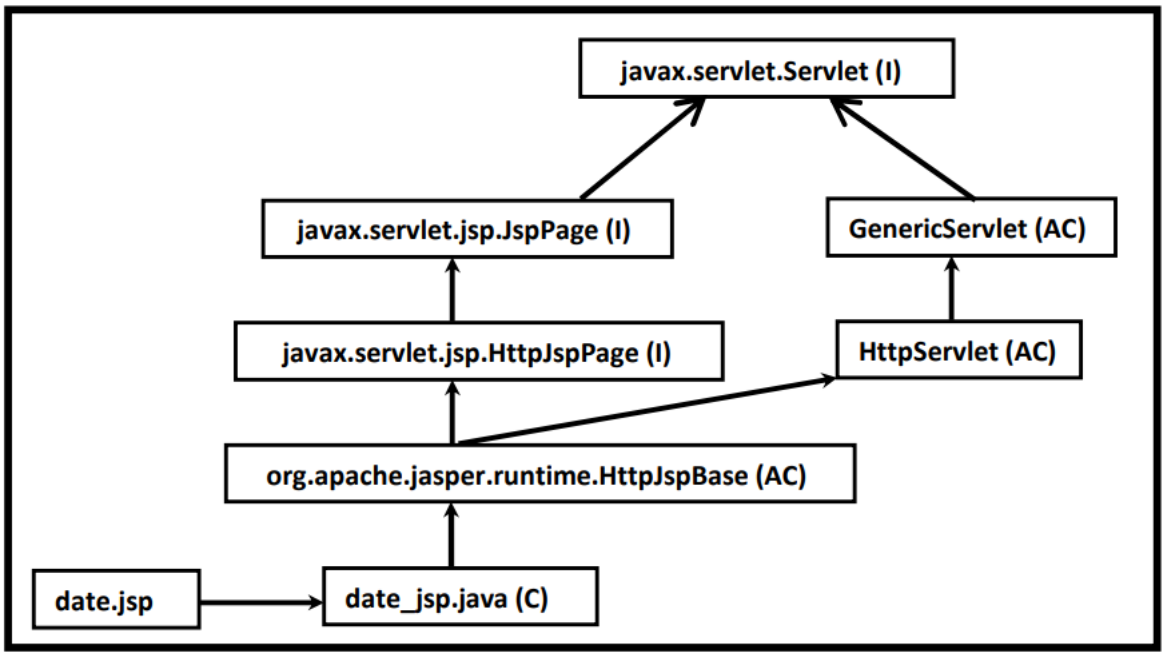
1. init()
2. jspInit()
3. destroy()
4. jspDestroy()
5. service()
6. _jspService()

1,3,5 are final methods in parent class
6th method automatically generated by JSP Engine.

JSP LifeCycle
==========







JSP will be converted to Servlet and that Servlet will be executed by the container to generate the response.

welcome.jsp ------translated to servlet by
jasper container(jasper.jar/jsp-api.jar)------> welcome_jsp.java

Translation Phase

1. JSP loading
2. JSP parsing
3. Converting JSP to Servlet
4. Compilation JSP to Servlet(.class)

Request Processing Phase

1. Servlet Loading
2. Servlet Instantiation
3. Servlet Initialization
4. Request processing
5. Servlet DeInstantitaion

Behind the Scenes

welcome_jsp.java

1.initalisation

```
jspInit(){

}
```

2.RequestProcessing

```
_jspService(HttpServletRequest request,HttpServletResponse response){
        .....
        .....
        .....
}
```

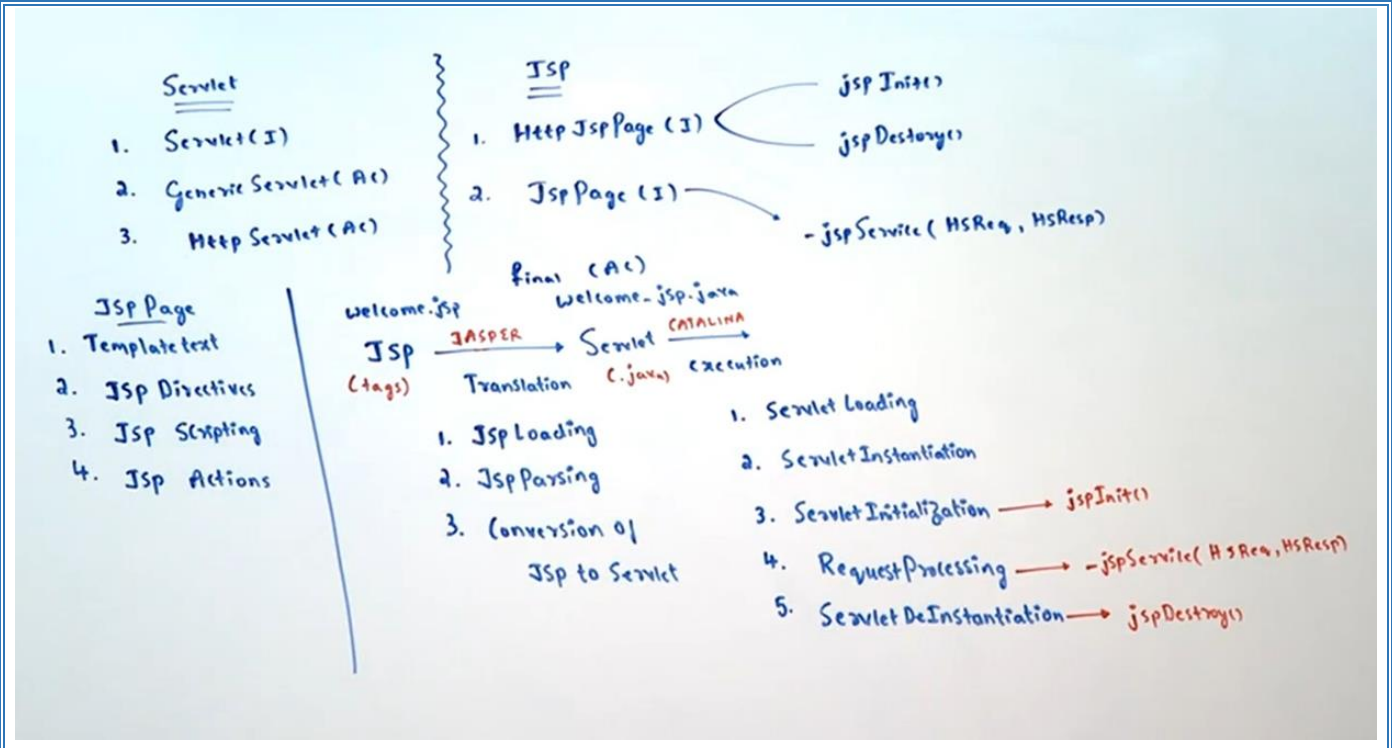
3.DeInstantiation

```
destroy(){
        .....
        .....
        .....
        .....
}
```

***Note:

JSP will participate in Translation Phase in the following cases

1. At the time of First Request
2. If the source code of JSP got modified when compared with earlier requests. For this JSP Engine uses ARAXIS Tool to compare time stamps of .class and .jsp files.



Pre Compilation of JSP:

In the case of JSP, at the time of first request the following activities will be performed.
1. Translation
2. Compilation
3. Class Loading
4. Instantiation
5. jspInit()
6. _jspService()

But for second request onwards only _jspService() method will be executed. Hence the processing time of first request is more when compared with remaining requests. To overcome this problem we should go for pre compilation of JSP.
We can invoke pre compilation as follows..

http://localhost:9999/jspdapp1/demo.jsp?jsp_precompile=true

It is not request to jsp and it is just performing pre compilation. B'z of this the following activities performed
1.Translation
2.Compilation
3.class Loading
4.Instantiation
5.jspInit()

Whenever we are sending first request, only _jspService() method will be executed.

http://localhost:9999/jspdapp1/demo.jsp

Hence the main advantage of Pre compilation is all requests will be processed with uniform response time.

JSP(JAVA SERVER PAGES)

==========

Note:

JspPage

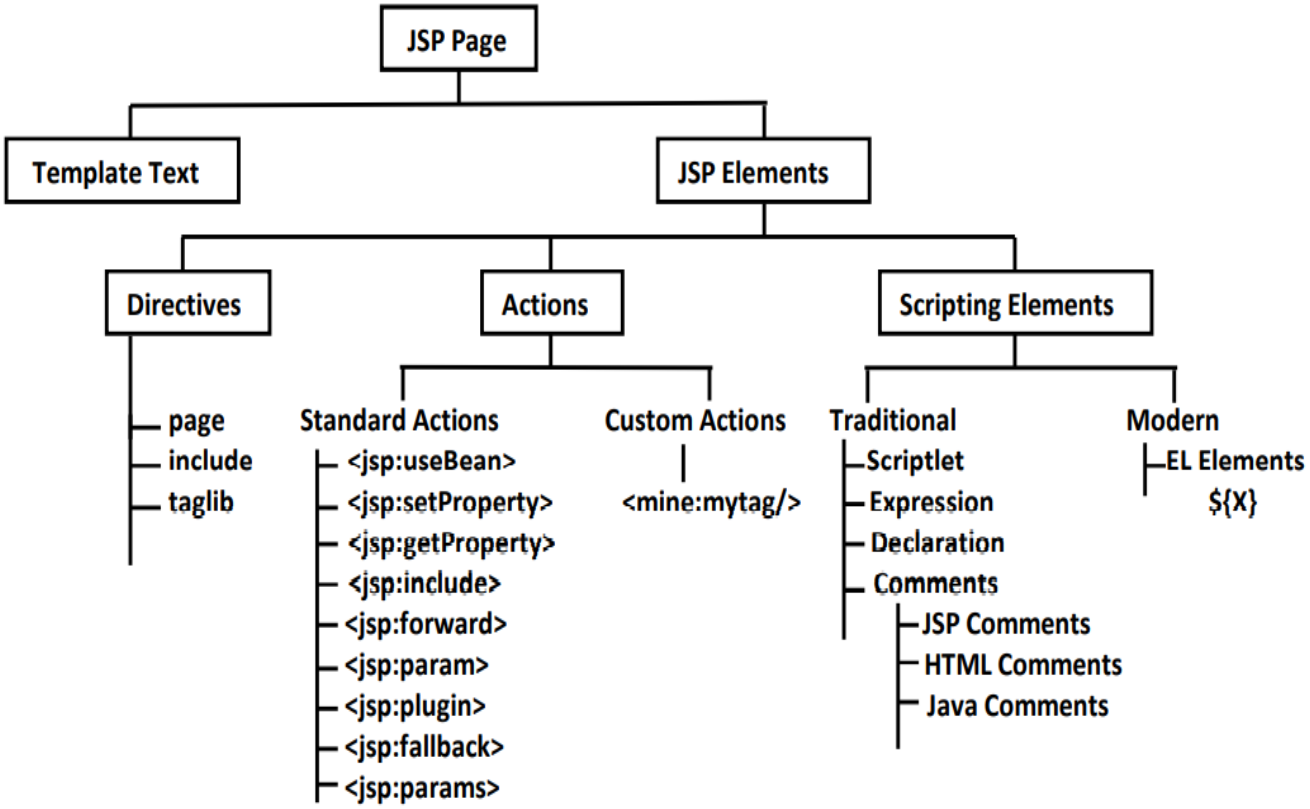
1. jspInit()
2. jspDestroy()

HttpJspPage

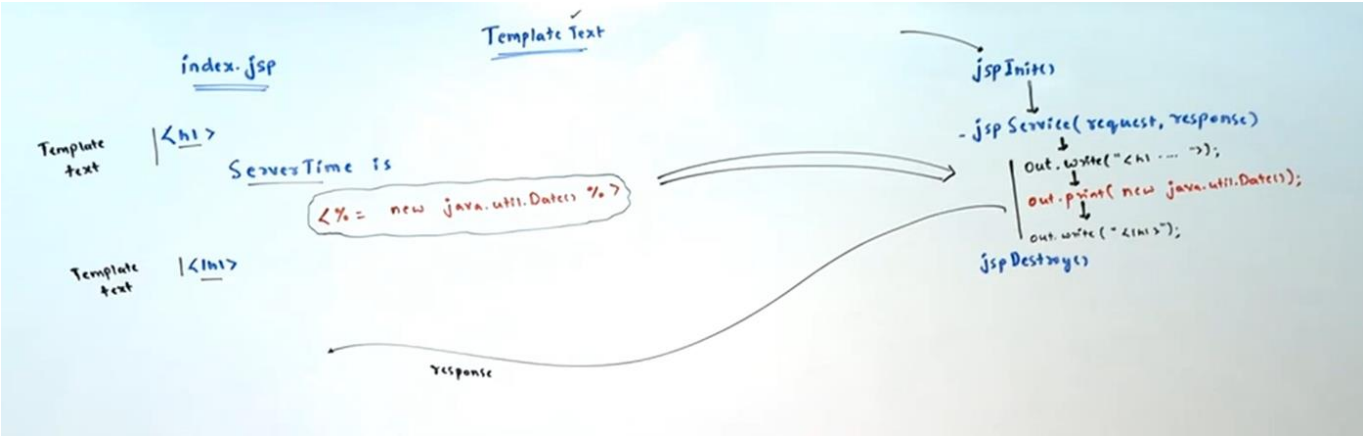
1. _jspService(request,response)

JSPElements

==========



Template Text
==========
It contains plain text data and html,xml tags.
For the template text no processing is required and it will become argument to out.write() method
in _jspService() method.



test.jsp:
<h1>The Server Time is:<%= new java.util.Date() %></h1>

For the above jsp the generated servlet class is:
public final class demo_jsp extends... {
public void _jspService(..)... {
out.write("<h1>The Server Time is:");
out.print(new java.util.Date());
out.write("</h1>");
}
}

Q. Template text will become argument to write() method where as expression value will become argument to print() method. What is the reason?

=> write() method can take only character data as argument
=> print() method can take any type of data as argument
=> Template text is always character data and hence it will become argument to write() method.
=> But expression value can be any type and hence we should not use write() method, compulsory we should go for print() method which can take any type of argument.

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 available user defined tag library into the present Jsp page.
   In web applications, Jsp Scripting Elements can be used to provide java 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?

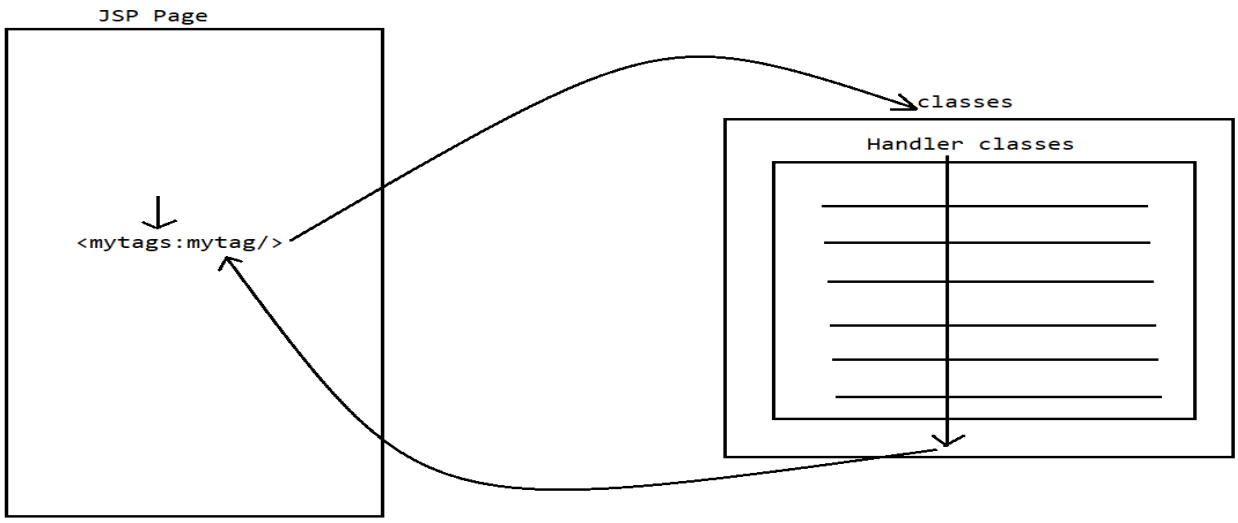
ANS:

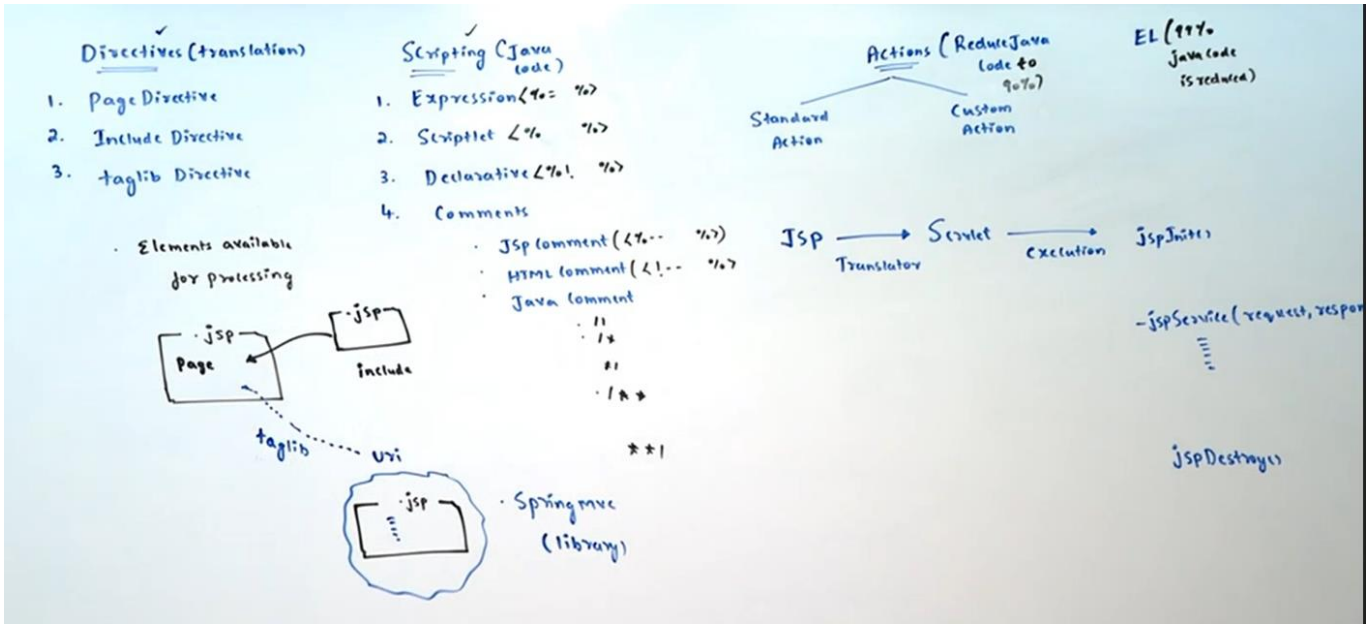
In Jsp applications, Scripting Elements can be used to allow java code inside Jsp pages but the main theme of Jsp technology is not to allow java code inside the Jsp pages.

In the above context, to preserve the theme of Jsp 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

refer:JspActions.png





Page directive:

=> page directive specifies overall properties of JSP Page to the JSP Engine.
like Tell me about yourself? in HR Round)

Eg1: <%@ page language="java" %>

This page directive specifies to the JSP Engine that the scripting language used in this jsp is java.

Eg2: <%@ page session="true" %>

syntax: <%@ page [Attributelist = Attributevalue]

This page directive specifies that the current jsp participating in session management.

=> We can use page directive any number of times, anywhere in the jsp.

=> The following is the list of all possible 13 attributes of page directive.

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

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:
We can use import attribute to import classes and interfaces of a particular package in the JSP.
This is similar to core java import statement.

demo.jsp
========
without import attribute:

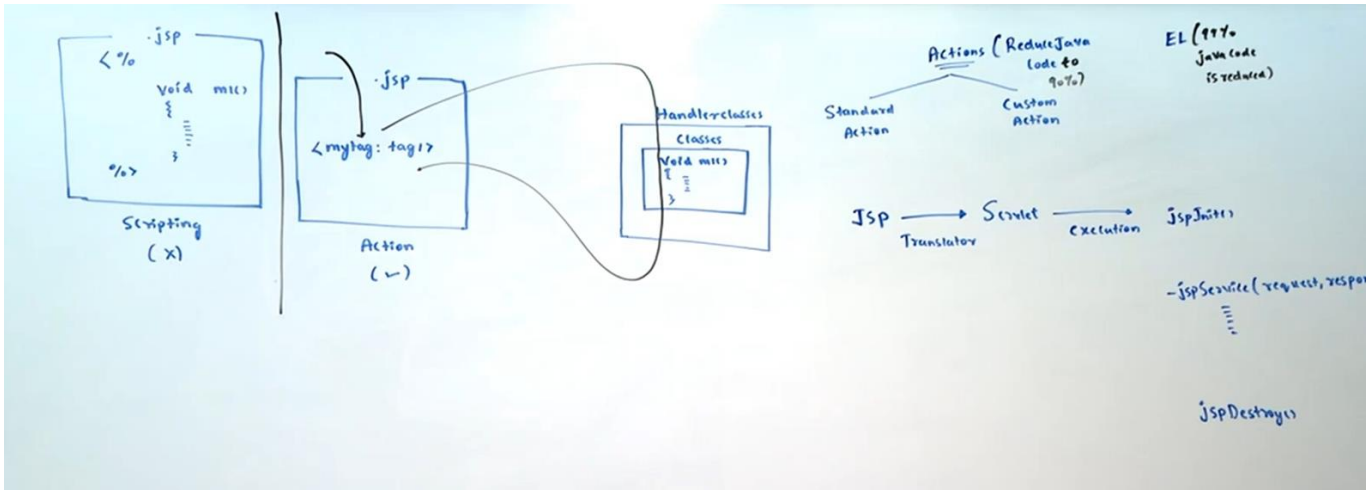
```
<h1>The Server Time is:<%= new java.util.Date() %></h1>
```

demo.jsp
========
with import attribute:

```
<%@ page import="java.util.Date" %>
<h1>The Server Time is:<%= new Date() %></h1>
```

To import multiple packages the following are various possibilities...
1) <%@ page import="java.util.*" %>
   <%@ page import="java.io.*" %>
2) <%@ page import="java.util.*" import="java.io.*" %>
   <%@ page import="java.util.*,java.io.*" %>
3) <%@ page import="java.util.*" import="java.io.*" %>

*****Note:**
Within the same JSP, we are not allowed to take any attribute multiple times with multiple values. But we can take multiple times with same value. This rule is not applicable for import attribute.



Q. Which of the following page directives are valid?
1) <%@ page session="true" session="false" %>//invalid
2) <%@ page session="true" session="true" %>//valid
3) <%@ page session="true" %> //valid
4) <%@ page import="java.util.*" import="java.io.*" %> //valid
5) <%@ page import=java.util.* %> //valid

Note:
Inside JSP, the following packages are by default available and hence we are not required to import these packages explicitly.

=> java.lang.*;
|=> System,String,Runnable,Cloneable,..…

=> javax.servlet.*;
|=> Servlet(I),GenericServlet(AC)
|=> ServletConfig(I),ServletContext(I)
|=> ServletRequest(I),ServletResponse(I)

=> javax.servlet.http.*;
|=> HttpServlet(AC)
|=> HttpServletRequest(I),HttpServletResponse(I)
|=> HttpSession(I)
|=> Cookie(C)

=> javax.servlet.jsp.*;
|=> JspPage(I)
|=> HttpJspPage(I)

4.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.

eg: <%= getServletInfo()%>

5. EXTENDS:

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

EX: <% @page extends=”com.abc.controller.MyServlet”%>

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.

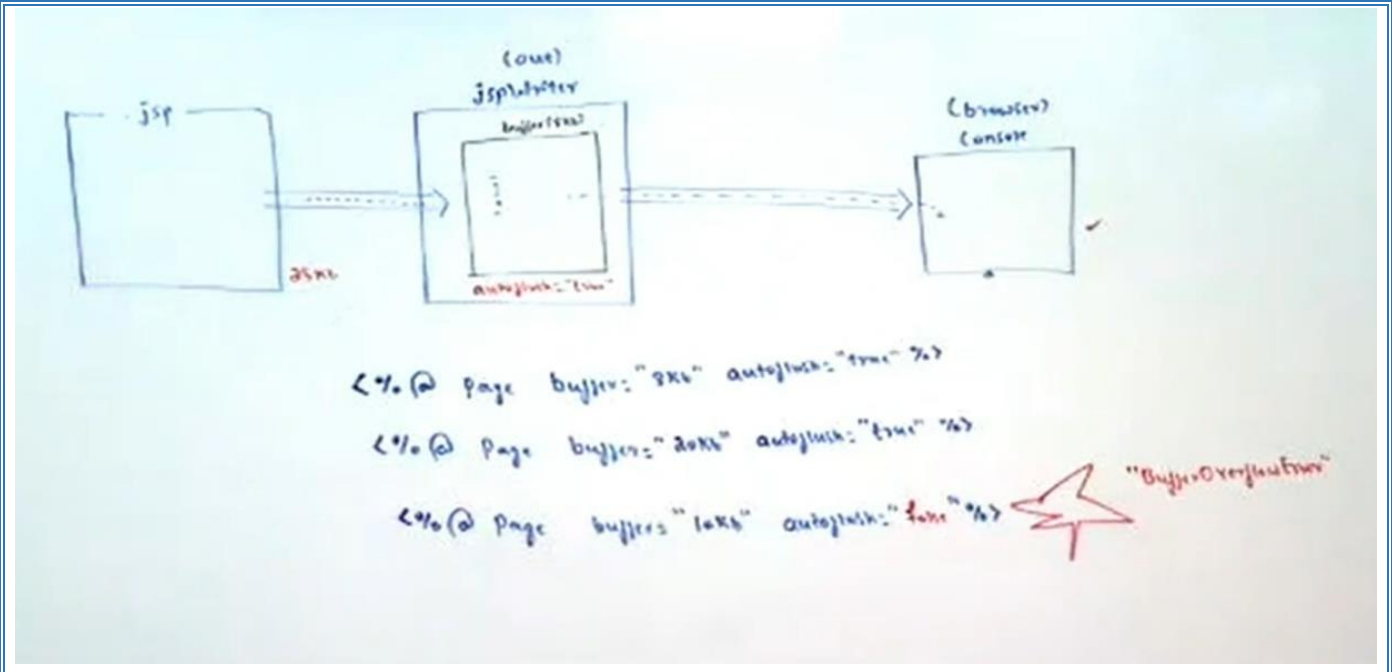
6. BUFFER:
This attribute can be used to specify the particular size to the buffer available in JspWriter object.
NOTE: Jsp technology is having its own writer object to track the generated dynamic response, JspWriter will provide very good performance when compared with PrintWriter in servlets.

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.

Summary of Page Attributes

Attribute	Purpose	Default Value
1) import	To Import Classes and Interfaces	No Default Value. But the following 4 Packages are not required to import because available by Default in every JSP. 1. java.lang 2. javax.servlet 3. javax.servlet.http 4. javax.servlet.jsp
2) session	To make Session Object unavailable to the JSP	true
3) contentType	To specify MIME Type of Response	text/ html
4) isELIgnored	To disable Expression Language in the JSP	false
5) isThreadSafe	To provide Thread Safety to the JSP	true



eg:

```
<%@ page buffer="52kb" autoFlush="true"%>
<%
        for(int i=0;i<=1000000;i++)
                out.println("sachin");
%>
```

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 like
org.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>
 <%= exception%>
 </font></b></center></body>
</html>
```

10. session:

By default session object is available in every JSP. If we don't want session object then we can make it unavailable by using page directive session attribute as follows

<%@ page session="false" %>

If we are not declaring session attribute explicitly or declaring session attribute with "true" value, then the generated servlet code is:

```
HttpSession session = null;
session = pageContext.getSession();
```

If we declare explicitly with false value, then the above code won't be generated.
The allowed values for the session attribute are:
true
TRUE
True
FALse
i.e. case insensitive String of true or false is allowed.

Note:
If we are taking any other value then we will get Translation Time error.
Eg: <%@ page session="nitin" %>
Error: invalid value for session

eg:

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

<%
        session.setAttribute("A", "sachin");
        session.setAttribute("B", "dhoni");
%>
<%= session.getAttribute("A") %>
<%= session.getAttribute("B") %>
```

11. isThreadSafe:
By Default a single JSP page can be accessed by multiple threads simultaneously. Hence JSP is not thread safe by default.

To provide Thread Safety to the JSP we should go for isThreadSafe attribute.
1) isThreadSafe="true":
It means JSP page is already thread safe & it is not required generated servlet to implement SingleThreadModel.
In this case JSP Page can process any number of client requests simultaneously.

2) isThreadSafe="false":
The current JSP is not thread safe. Hence to provide thread safety, Generated servlet will implements SingleThreadModel interface.
In this case JSP Page can process only one request at a time.

Note: The default value of isThreadSafe attribute is true.

Q. To provide Thread Safety, which of the following arrangement we have to take in the JSP?

1. <%@ page isThreadSafe="true" %>
2. <%@ page isThreadSafe="false" %>
3. Not required any arrangement b'z every JSP is by default Thread Safe

ans. 2

11. isELIgnored:

Expression Language(EL) has introduced in JSP 1.2V

The main objective of EL is to remove java code from the JSP.

1. isELIgnored="true"

EL Syntax won't be processed and just treated as plain text.

2. isELIgnored="false":

EL Syntax will be processed and print its value.

Eg: http://localhost:7777/jspdapp1/demo.jsp?user=Mallika

demo.jsp:

```
<%@ page isELIgnored="false" %>
<h1>User Name: ${param.user}</h1>
```

o/p: User Name: Mallika

JspDay3:-

JSP(JAVA SERVER PAGES)

==========

Note:

JspPage

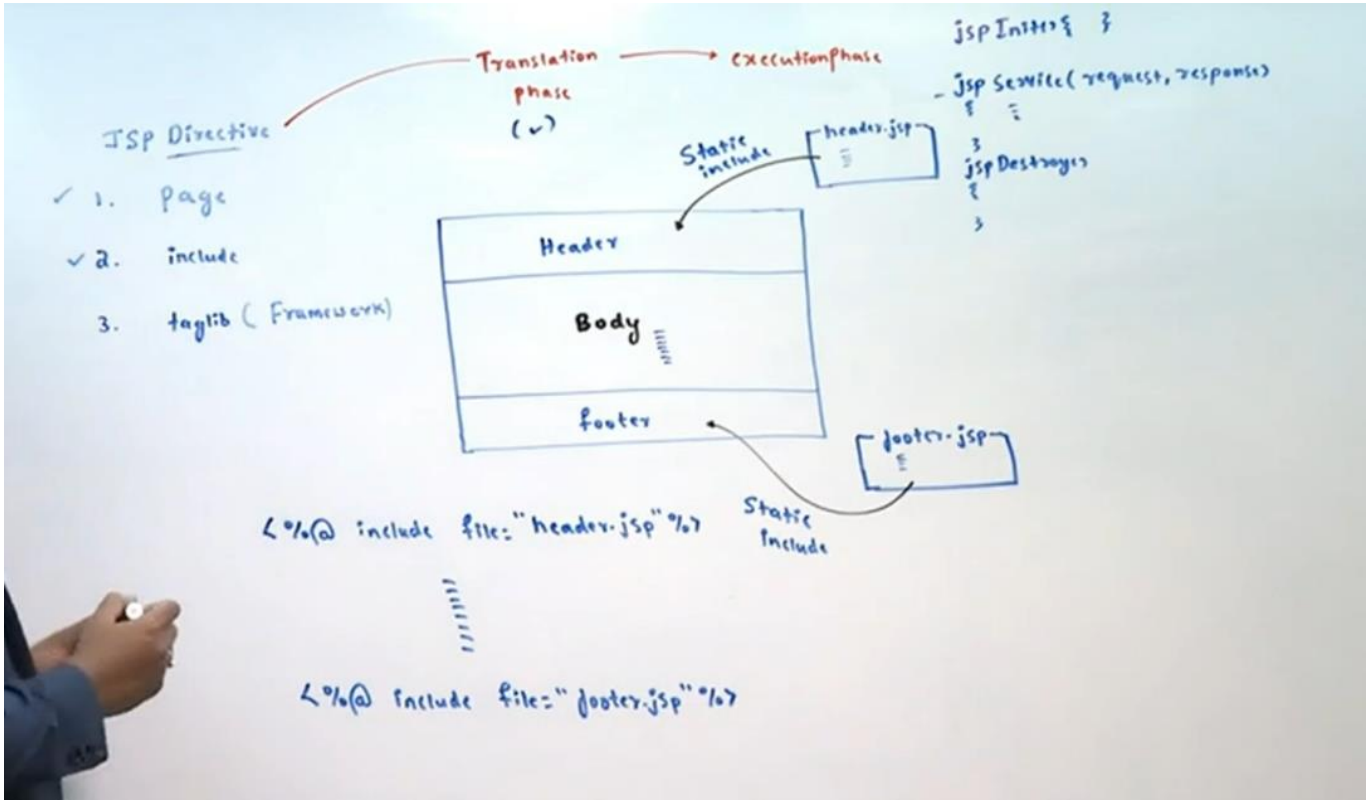
1. jspInit()
2. jspDestroy()

HttpJspPage

1. _jspService(request,response)

include directive:

syntax: <%@ include file='[any resource]' %>



=> If several JSPs contain same code then it is recommended to separate that common code in a separate file. Wherever that common code is required just we have to include that file.

=> This mechanism is called inclusion mechanism and we can use very commonly to include header and footer information which is common in every JSP.

The main advantages of Include mechanism are:
1) It promotes code reusability
2) It improves maintainability of the code
3) Enhancement will become very easy

=> We can implement include mechanism either by include directive or by include action.

Include Directive:

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

The content of second.jsp will be included in the current jsp at translation time. Hence this inclusion is also known as static include or translation time inclusion.

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

first.jsp

```
<%@ include file="second.jsp" %>
<h1>This is First JSP</h1>
```

second.jsp:

```
<h1>This is Second JSP</h1>
```

For both including and included JSPs, a single servlet will be generated.

Include Action:

```
<jsp:include page="second.jsp" flush="true" />
```

The response of second.jsp will be included in the current page response at runtime/request processing time.Hence this inclusion is also known as Dynamic include or Runtime inclusion.

first.jsp

```
1. <jsp:include page="second.jsp" flush='true'/>
2. <h1>This is First JSP</h1>
```

second.jsp:

```
<h1>This is Second JSP</h1>
```

Note:
For both including and Included JSPs, separates servlets will be generated.

Differences b/w Include Directive and Include Action

Include Directive
==========

1) <%@ include file = "second.jsp" %>
Contains only one Attribute File.

2) The Content of Target JSP will be included at Translation Time.
Hence it is also considered as Static Include.

3) For both including and included JSPs, a Single Servlet will be generated. Hence Code sharing between the Components is possible.

4) Relatively Performance is High.

5) There is no Guarantee for Inclusion of latest Version included JSP. It is Vendor Dependent.

6) If the Target Resource won't change frequently then it is recommended to use Static Include.

Include Action

1) <jsp:include page = "second.jsp" flush = "true"/>
Contains 2 Attributes Page and File.
2) The Response Target JSP will be included at Runtime. Hence it is also considered as Dynamic Include.
3) For both including and included JSPs, separate Servlets will be generated. Hence code sharing between the Components is not possible.
4) Relatively Performance is Low.

5) Always latest Version of included Page will be included.

6) If the Target Resource will change frequently then it is recommended to use Dynamic Include

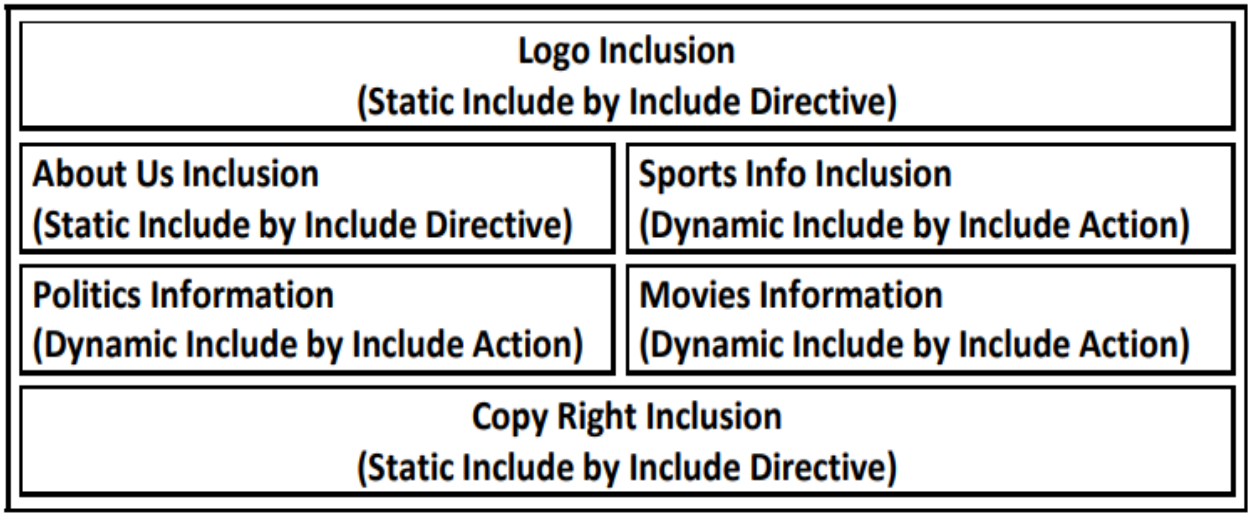
Note:
To include the content of header.jsp at translation time, the required import is:
<%@ include file="header.jsp" %>

To include the response of header.jsp at request processing time (run time), the required import
is: <jsp:include page="header.jsp" flush="true"/>

Q. Which of the following inclusions are valid?

1.<%@ include page="header.jsp" %>//invalid
2.<%@ include file="header.jsp" %> //valid
3.<jsp:include file="header.jsp" /> //invalid
4.<jsp:include page="header.jsp" /> //valid(flush is default)
5.<jsp:include page="header.jsp" flush="true"/>//valid
6.<%@ include file="header.jsp" flush="true"%>//invalid

Note:
page attribute is applicable for include action where as file attribute is applicable for include directive.
flush attribute is applicable only for include action but not for include directive.
refer: requirement.png



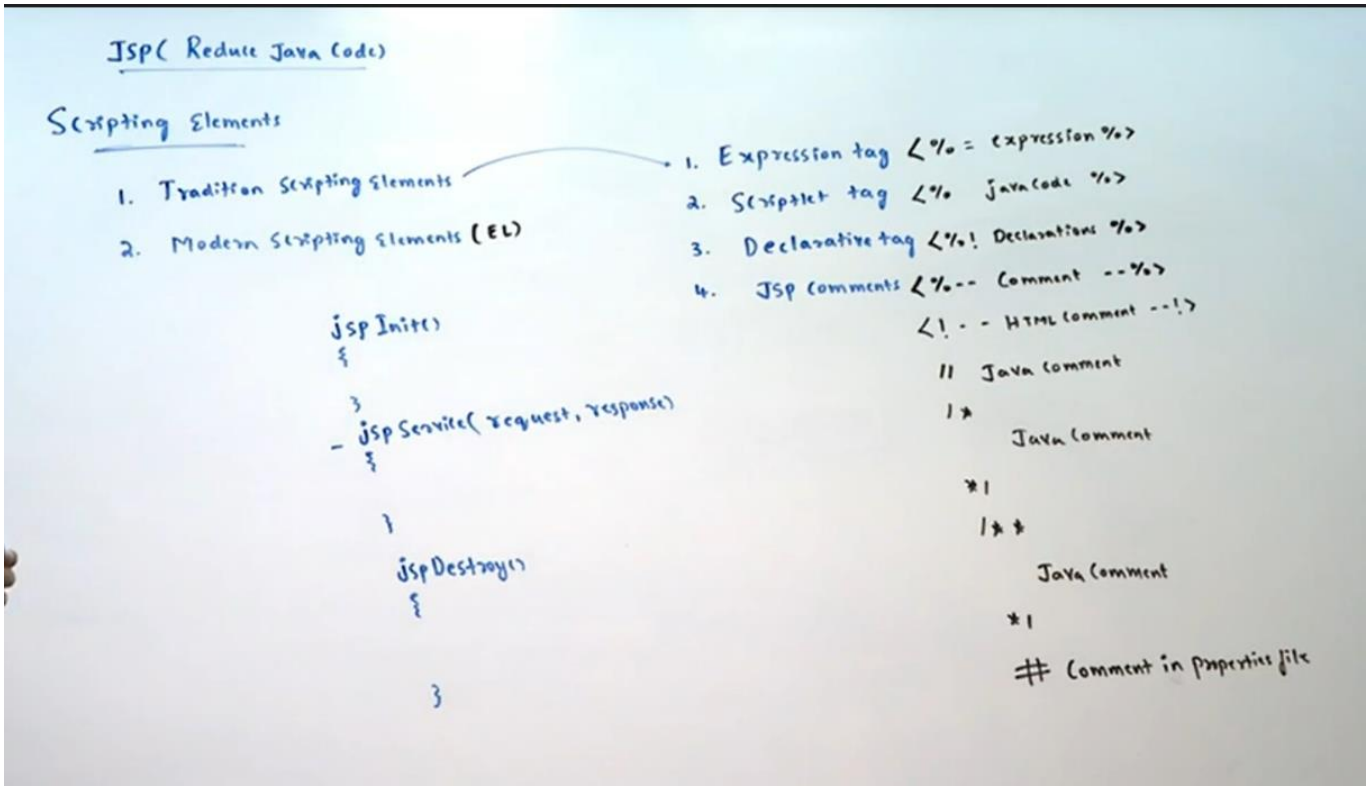
3.taglib directive(Specifically used in SpringMVC framework to generate the form using Spring supplied library)

=> We can use taglib directive to make custom tags available to our jsp.
<%@ taglib prefix="mine" uri="/WEB-INF/MyTld.tld" %>
=> taglib directive contains 2 attributes prefix and uri.
=> uri represents the location of TLD File which in turn represents TagHandler class.

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
4. Jsp Comment



1. DECLARATIONS:

We can use Declaration tag to declare instance variables, static variables, instance blocks, static blocks, methods etc..

Syntax:

```
<%!
        Any java declarations
 %>
```

These declarations will be placed directly in the generated servlet but outside of _jspService() method.

```
<%!
        int x =10;
        static int y = 20;
        int[] a ={10,20,30};
        public void m1() { }
%>
```

Every Java statement inside Declaration tag should compulsary ends with semicolon(;)

demo.jsp:

```
 <%!
   public void m1() {
         out.println("Hello");
   }
  %>
```

CE: out cannot be resolved.

*****Note:**

All JSP Implicit objects are declared as local variables of _jspService() method.But declaration tagcode will be placed outside of _jspService() method. Hence we cannot use JSP implicit objects inside Declaration tag.

2. Scriptlet:

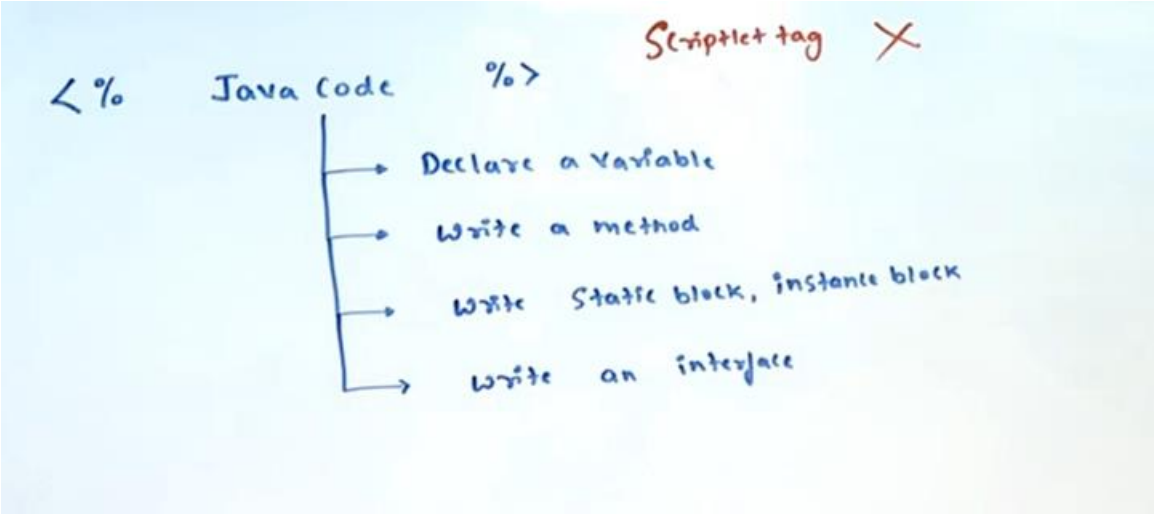
We can use scriptlet to place java code in the jsp.

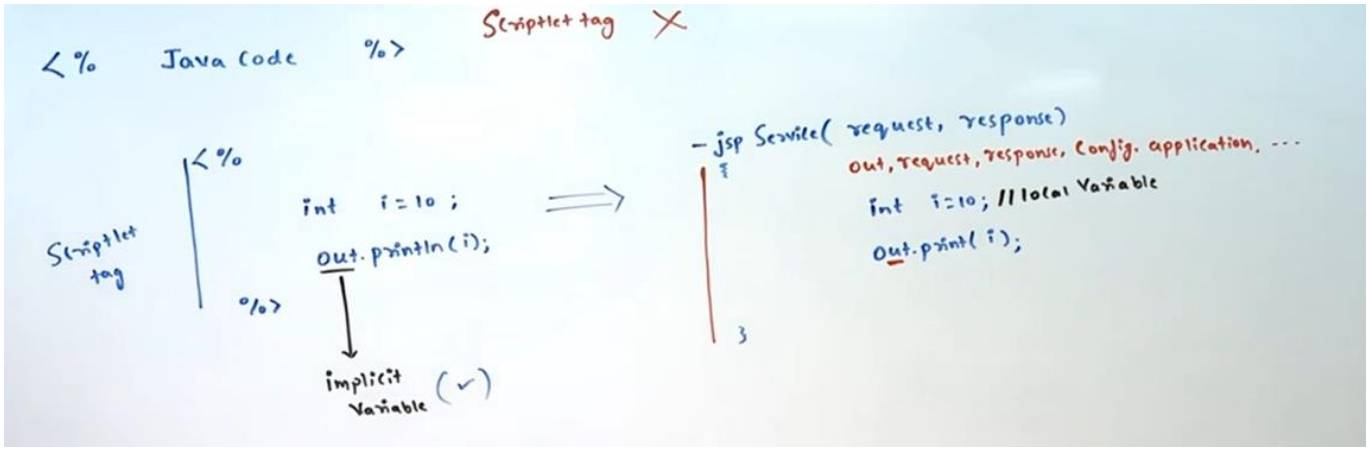
Syntax:

```
<%
   Any Java Code
%>
```

Java Code inside scriptlet will be placed directly inside _jspService() method of generated servlet.

Every java statement present inside scriptlet should compulsary ends with ;



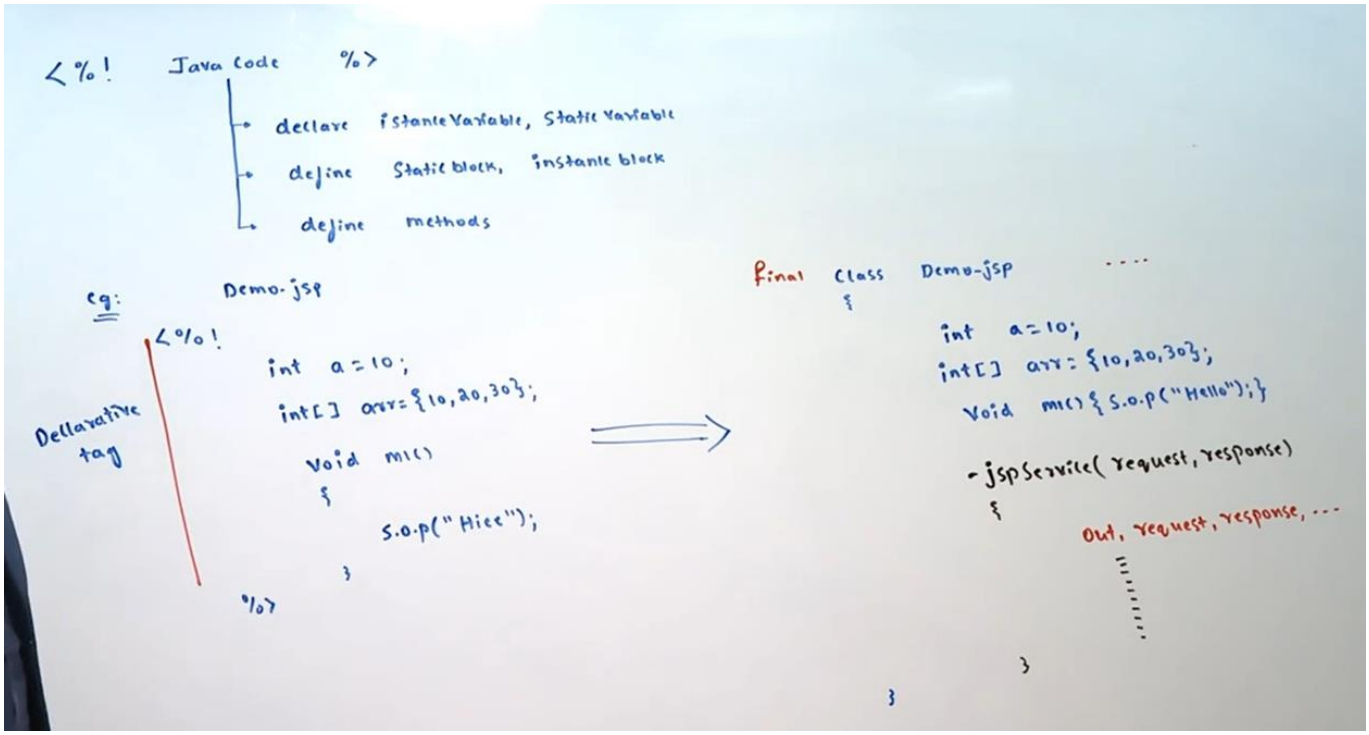


Write a JSP Print hit count of the JSP?

1.

```
<%
    int i=0;
    i++;
    out.println(i);
%>
```

here i is a local variable, so every time the value will be 1.



2.

```
<%!
        int i=0;
%>
<%
        i++;
        out.println(i);
%>
```

here i is instance variable for the generated servlet,so the value depends on the the user hit.

Q. What is the difference between the following?

1.

```
<%
        int x=10;
%>
```

2.

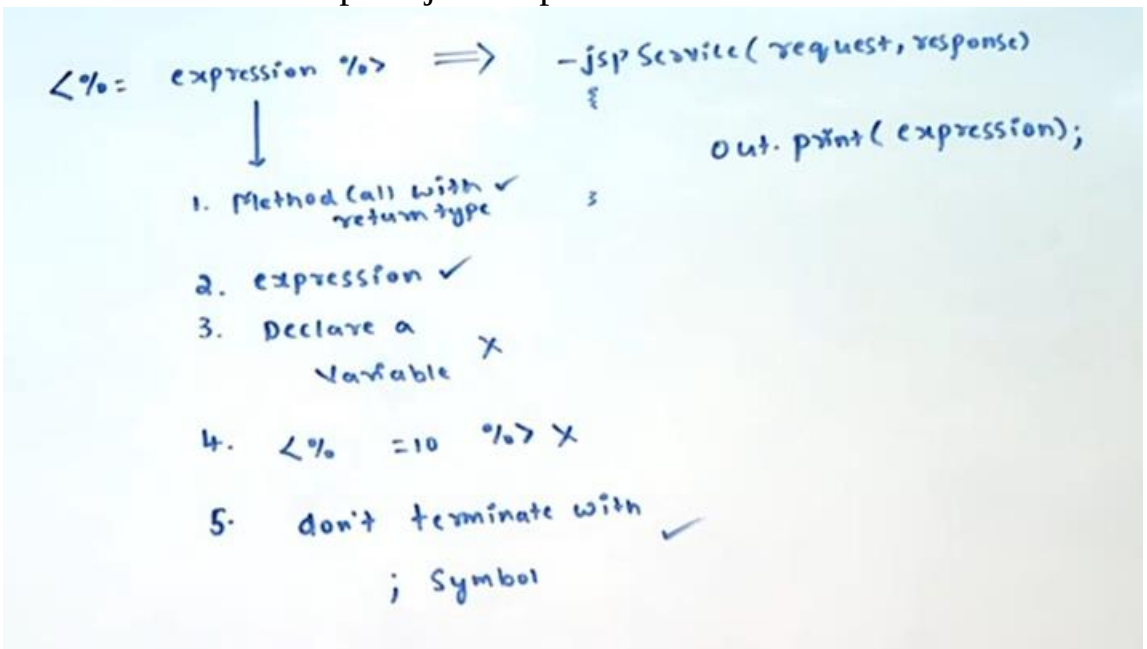
```
<%!
        int x=10;
%>
```

In the first case x is local variable of _jspService() method where as in the second case x is instance variable of generated servlet.

Note: It is not recommended to use scriptlets in the JSP

3. Expression:

Expression can be used to print java expression values to the JSP.



Syntax: <%= x %>
The equivalent generated servlet code is:

```
_jspService(..)...{
        out.print(x);
}
```

ie expression value will become argument to print() method inside _jspService() method.

Eg: test.jsp

```
UserName:<%= request.getParameter("user") %><br>
Password:<%= request.getParameter("pwd") %>
```

http://localhost:7777/jspdapp1/test.jsp?user=sachin&pwd=tendulkar

Conclusions:

1. Inside expression we are not allowed to use semi colon(;) otherwise we will get 500 error code.

```
        Eg: <%= new java.util.Date(); %>
        out.print(new java.util.Date(););
```

2. Inside expression we can use method calls also, but void return type method calls are not allowed

Eg:

```
<%= new java.util.ArrayList().size() %>
<%= new java.util.ArrayList().clear() %>
```

3. Inside expression space is not allowed between % and =, otherwise it is treated as scriptlet & it will be invalid.

Eg: <% =10 %> invalid

the equivalent generated servlet code is:

```
_jspService(){
        =10
}
```

4. Inside JSP Expression we are not allowed to use declarations

```
<%= String s ="nitin" %> //invalid
```

Q.Which of the following are valid java expressions?

1) <%= 27 %> //valid
2) <%= "27" %> //valid
3) <%= Math.random() %>//valid
4) <%= 10*20 %>//valid
5) <%= 10>20 %>//valid
6) <%= new Student() %>//valid
7) <%= String s ="sachin" %> //invalid

Output
1 to 6 valid,only 7 is invalid

Comments:

In the JSP, 3 types of comments are allowed.
1. JSP Comments
2. HTML Comments
3. Java Comments

1. JSP Comments:

<%-- This is JSP Comment --%>
These comments are visible only in the JSP and not visible in the remaining phases of JSP
Execution. Hence these comments also known as Hidden Comments.
It is highly recommended to use JSP Comments in the JSP.

2. HTML Comments:

<!-- This is HTML Comment -->
Also known as Template text comments.
These are visible to the end user as the part of generated response source code.
Hence these are not recommended to use in the JSP.

3. Java Comments:

```
// single line java comment

/*
        multiline java comment
*/
```

/**
Java Doc comment
*/
Also known as scripting comments.
These are also visible in the generated servlet source code but not visible in the remaining phases.

Note:
Among Expressions, Scriptlets, Declarations and JSP Comments, we cannot use one inside another.
i.e. Nesting of these scripting elements is not possible, otherwise we will get Compile Time Error.
refer summarypage.png

Summery of JSP Comments

Comment Type	Is it Visible in JSP	Is it Visible in Generated Servlet	Is it Visible in End User's Response - Source Code
1) JSP Comments	✔	X	X
2) HTML Comments	✔	✔	✔
3) Java Comments	✔	✔	X

Comparison Table of JSP Scripting Elements

Element	Syntax	Ends with ;	Is Code generated inside _jspService() Method OR not?
Expression	<%= X %>	No	Yes
Scriptlet	<% Any Java Code %>	Yes	Yes
Declaration	<%! Any Java Declarations %>	Yes	No
Comments	<%-- This is JSP Comment %>	NA	NA

date.jsp
========
```
<%@page import="java.util.*"%>

<%--Declarative Tag --%>
<%!
        Date d=null;
        String date=null;
%>

<%-- Scriptlet Tag--%>
```

```
<%
        d=new Date();
        date=d.toString();
%>

<html>
<body bgcolor="lightyellow">
<center><b><font size="6" color="red"><br><br>
Today Date :<%-- Expression Tag --%> <%=date%>
</font></b></center></body>
</html>
```

Translated Servlet

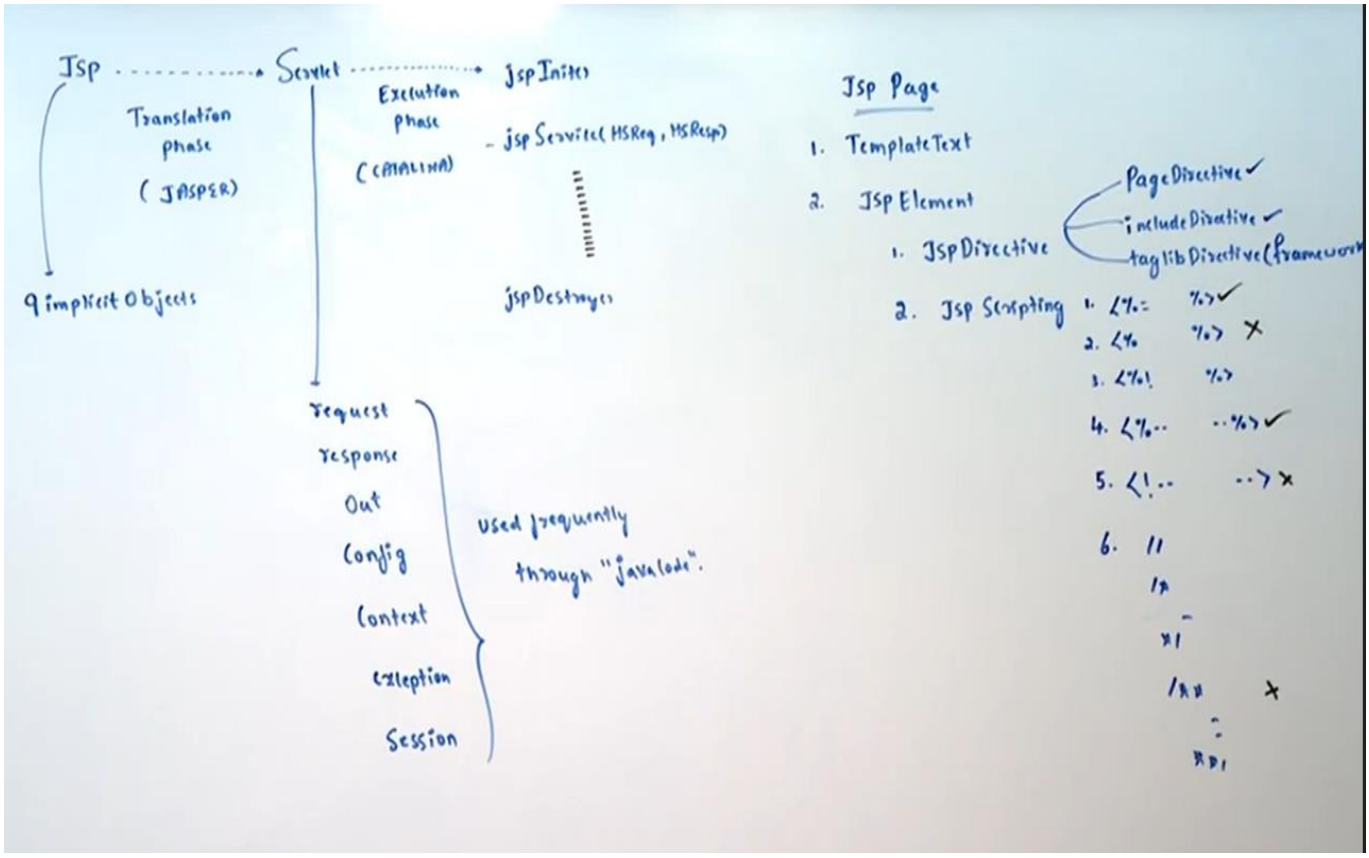
==========

```
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>");
 }
}
```

JspDay4:-

JSP(JAVA SERVER PAGES)
==========
Note:
JspPage
1. jspInit()
2. jspDestroy()

HttpJspPage
1. _jspService(request,response)



JSP Implicit Objects
==========

Objective:
For the given design goal write the JSP Code using appropriate implicit objects.?
When compared with Servlet Programming, Developing JSPs is very easy because the required mandatory stuff automatically available in every JSP. Implicit objects also one such area ,which are by default available for every JSP.

The following is the list of all possible 9 JSP implicit objects.

1) request ➔ HttpServletRequest(I)
2) response ➔ HttpServletResponse(I)
3) config ➔ ServletConfig(I)
4) application ➔ ServletContext(I)
5) session ➔ HttpSession(I)
6) out ➔ javax.serlvet.jsp.JspWriter(AC)
7) page ➔ java.lang.Object(CC)
8) pageContext ➔ javax.servlet.jsp.PageContext(AC)
9) exception ➔ java.lang.Throwable(CC)

1) Request and Response Implicit Objects:

Request and Response Implicit Objects are available as arguments to service() method.
All the methods of HttpServletRequest and HttpServletResponse can be applicable on these request and response objects.

request.jsp

==========

```
<h1>The Request Method:<%= request.getMethod() %><br>
   User Name: <%= request.getParameter("user") %><br>
   Client IP Address: <%= request.getRemoteAddr()%><br>
   Content Type:<%= response.getContentType() %>
</h1>
```

http://localhost:7777/jsp1/request.jsp?user=sachin

2) Application Implicit Object:

This implicit object is of type ServletContext,which represent the environment of application.
Whatever methods we can apply on ServletContext object,we can apply all those methods on application implicit object.

application.jsp

==========

```
<h1>
   The context parameter User Name: <%=
  application.getInitParameter("uname") %><br>
   The Application Name:<%= application.getServletContextName() %>
</h1>
```

web.xml:

```
<web-app>
   <display-name>JSP Implicit Objects Application</display-name>
   <context-param>
      <param-name>uname</param-name>
      <param-value>scott</param-value>
   </context-param>
</web-app>
```

http://localhost:7777/jsp1/application.jsp

3) Session Implicit Object:

In every JSP session implicit object is by default available and it is of type HttpSession.

Whatever methods we can apply on HttpSession object, all those methods we can apply on session implicit object.

session.jsp

==========

```
<h1>
   The Session ID is : <%= session.getId() %> <br>
   The Session Time out is : <%= session.getMaxInactiveInterval() %> Seconds <br>
   Is the session is newly created: <%= session.isNew() %> <br>
</h1>
```

If we can make session implicit object unavailable by using page directive then we are not allowed to use session implicit object. otherwise we will get compile time error.

test.jsp

========

```
<%@ page session="false" %>
 The Session ID is : <%= session.getId() %>
```

CE: session cannot be resolved.

Q. In which of the following cases session object available in the JSP?

1. <%@ page session="true" %>
2. <%@ page session="false" %>
3. <%@ page session="true" session="false" %>
4. <%@ page contentType="text/html" %>

1,4 available,remaining all cases it is false.

4) Config Implicit Object:

config object is of type ServletConfig.
Whatever methods we can apply on the ServletConfig object,we can apply all those methods on config also.

The following is the list of all applicable methods.

1) getServletName()
2) getInitParameter(String pname)
3) getInitParameterNames()
4) getServletContext()

config.jsp

==========

```
 The Logical Name: <%= config.getServletName() %><br>
 The Init Param value is :
<%= config.getInitParameter("hotTopic") %>
```

web.xml

=======

```
<web-app>
  <servlet>
      <servlet-name>DemoJSP</servlet-name>
      <jsp-file>/config.jsp</jsp-file>
```

```
        <init-param>
                <param-name>hotTopic</param-name>
                <param-value>IPL</param-value>
        </init-param>
  </servlet>
  <servlet-mapping>
        <servlet-name>DemoJSP</servlet-name>
        <url-pattern>/test</url-pattern>
  </servlet-mapping>
</web-app>
```

http://localhost:7777/jsp1/test
The Logical Name: DemoJSP
The Init Param value is : IPL

http://localhost:7777/jsp1/config.jsp
The Logical Name: jsp
The Init Param value is : null

Note:
To reflect Servlet level web.xml configurations in the JSP, compulsory we should access by using url-pattern.

5) pageContext implicit object

The pageContext implicit object is of type javax.servlet.jsp.PageContext.
It is abstract class and web server vendor is responsible to provide implementation.
PageContext is the child class of JspContext.

```
JspContext  ---------->  Simple Custom Tag  (JSP 2.0 V)
   |
   |
PageContext ---------->  Classic Custom Tag (JSP 1.1 V)
```

We can use pageContext implicit object for the following 3 purposes.

1) To get all other implicit objects
2) To Perform Request Dispatching activities(include and forward)
3) To perform attribute management in any scopes(dynamic data management)

a) Getting JSP Implicit objects from pageContext:
By using pageContext implicit object,we can get all other JSP implicit objects. i.e pageContext acts as single point of contact for all other implicit objects.

PageContext class contains the following methods for this.

1) request ➔ getRequest()
2) response ➔ getResponse()
3) config ➔ getServletConfig()
4) application ➔ getServletContext()
5) session ➔ getSession()
6) out ➔ getOut()
7) page ➔ getPage()
8) exception ➔ getException()

These methods are not useful within the JSP. We can use these methods outside of JSP, mostly in Custom Tag Handlers.

b) Request Dispatching by using pageContext:

We can perform request dispatching by using the following methods of PageContext.

1) public void forward(String target)
2) public void include(String target)

The target resource can be specified by either relative path or absolute path.

Eg:

first.jsp

```
=========
<h1>Hello This is First JSP</h1>
  <%
       pageContext.forward("second.jsp");
  %>
```

second.jsp

```
==========
<h1>Hello This is Second JSP</h1>
```

forward:
o/p: Hello This is Second JSP

include:
o/p:
Hello This is First JSP
Hello This is Second JSP

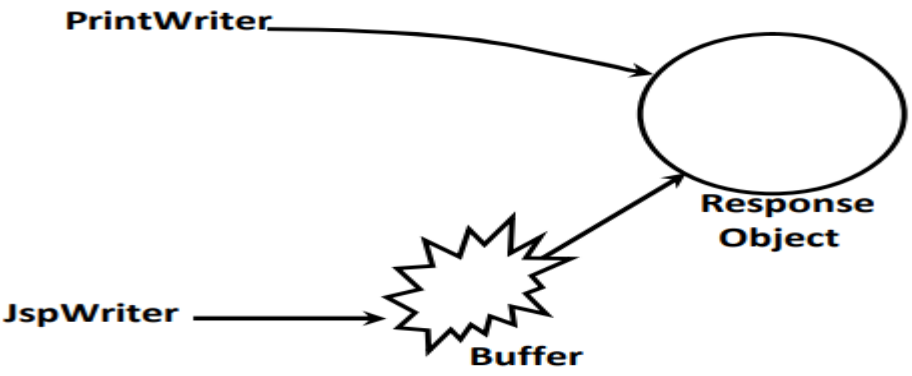
c) Attribute Management in any scope:

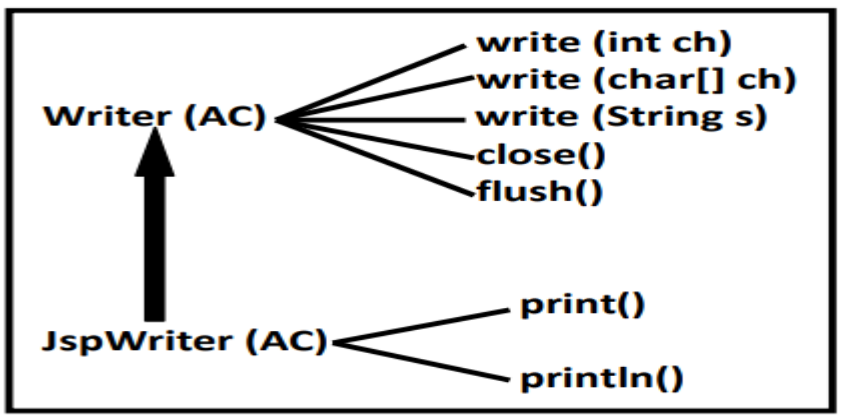
By using pageContext implicit object we can perform attribute management in any scope.(Will be covered in detail in the next topic: JSP Scopes)

6. out implicit object:

=> This implicit object is of type javax.servlet.jsp.JspWriter.
=> This class is specially designed for JSPs to write character data to the response.
=> The main difference b/w JspWriter and PrintWriter is, in PrintWriter buffering concept is not available. Hence if we are writing anything by using PrintWriter, it will write directly to the response object.
=> But in the case of JspWriter, buffer concept is available. Hence if we are writing anything by using JspWriter, first it will write to buffer and at least total buffered data will write to response.
=> Except this buffer difference there is no other difference b/w JspWriter and PrintWriter.

JspWriter = PrintWriter + Buffer





test.jsp:

```
<%@ page import="java.io.*" %>
<%
  PrintWriter pw=response.getWriter();
  out.print("<h1>sachin</h1>");
  pw.print("<h1>saurav</h1>");
  out.print("<h1>sachin</h1>");
  pw.print("<h1>saurav</h1>");
  out.print("<h1>sachin</h1>");
  pw.print("<h1>saurav</h1>");
%>
```

http://localhost:7777/jsp1/test.jsp

o/p:

sachin
sachin
sachin
saurav
saurav
saurav

Note:

Within the JSP we can use either PrintWriter or JspWriter but not recommended to use both simultaneously.

Note:

=> JspWriter is the child class of Writer(AC).Hence all methods of Writer class are by default available to the JspWriter through inheritance.

=> In addition to these methods, JspWriter contains its own set of print() and println() to add any type of Data to the response.

test.jsp:

```
<h1>
 <%
       out.print("The PI value is :");
       out.print(3.14);
       out.print("This is exactly ");
       out.print(true);
  %>
</h1>
```

Q1. In JSP, How we can write Response?
By using out implicit object, which is of the type : JspWriter

Q2. What is the difference b/w PrintWriter and JspWriter?
JspWriter = PrintWriter+Buffer;

7) Page Implicit Object:
The Page Implicit Object is always pointing to current servlet object.
In the generated servlet it is declared as follows..
Object page = this;

Note:
=> Parent reference can be used to hold child class object. This property is called polymorphism.
But by using parent reference, we cannot call child specific methods and we can call only methods present in Parent class.
=> Object page=this;
=> page variable is declared with Object type. Hence on page variable we can call only methods available in Object class and we cannot call servlet specific methods. If we are trying to call then we will get compile time error.

If we perform type casting then we can call servlet specific methods.
Q. Which of the following are valid?
1. <%= page.getServletInfo() %>
2. <%= this.getServletInfo() %>
3. <%= getServletInfo() %>
4. <%= ((HttpServlet)page).getServletInfo() %>

2,3,4 are valid calls.
1 invaild calls becoz calling on Object class.

Note:
On the page variable we cannot call servlet specific methods. Hence page implicit object is the most rarely used object in the JSP.

8) Configuring Error Pages in the JSP (Exception Implicit Object):
=> It is not recommended to display java specific error information to the end user directly. We have to convert java specific error information into end user understandable form. For this we have to configure error pages.

=> We can configure error pages in JSPs by using the following 2 approaches
1) Declarative Approach
2) Programmatic Approach

1) Declarative Approach:
We can configure Error pages according to a particular exception or according to error-code in web.xml as follows..

```
<web-app>
    <error-page>
        <exception-type>java.lang.ArithmeticException</exception-type>
        <location>/error.jsp</location>
    </error-page>
    <error-page>
        <error-code>404</error-code>
        <location>/error404.jsp</location>
    </error-page>
</web-app>
```

The error pages configured in this approach are applicable to the entire web application.

2) Programmatic Approach:
We can configure error page for a particular jsp by using errorPage attribute of page directive.

demo.jsp
========

```
<%@ page errorPage="error.jsp" %>
<h1> The Result is :<%= 10/0 %></h1>
```

In demo.jsp if any exception or error occurs then error.jsp is responsible to report this error.
This way of configuring error page is applicable for a particular JSP.
http://localhost:7777/jsp1/demo.jsp

We can declare a JSP as error page by using isErrorPage attribute of page directive.
In the error pages only exception implicit object is available.

error.jsp

```
=========
  <%@ page isErrorPage="true" %>
  <h1>currently we are facing some problems plz try after some time..The problem
        is:
        <%= exception %></h1>
```

If JSP is not declared as error page but if we are trying to access exception implicit object then we will get compile time error saying - "exception cannot be resolved".

Note: If we are sending the request to error page directly without any exception, then exception implicit object refers "null"

Which approach is recommended?

Declarative Approach is recommended b'z we can customize error pages based on exception type and error code.

Note:

1. All JSP Implicit objects are available as local variables of _jspService() method in the Generated Servlet. Hence inside _jspService() method only we can access implicit objects. But outside of _jspService() method we cannot access.

2. Scriptlet and expression tags code will be placed inside _jspService() method. Hence inside scriptlet and expression tags we can use jsp implicit objects.

3. But declaration tag code will be placed outside of _jspService() method in the generated servlet. Hence we cannot use jsp implicit objects inside declaration tag.

eg1:

```
<%
 out.println("Hello");//valid
%>
```

eg2:

```
<%!
 public void m1(){
    out.println("Hello");//invalid
 }
%
```

eg3:
Session Id: <%= session.getId() %>//valid

2. Among all JSP implicit objects except session and exception, all remaining objects always available in every JSP. We cannot make these objects unavailable.
session object by default available in every JSP. But we can make it unavailable by using page directive as follows..

<%@ page session="false" %>

exception implicit object is by default not available in every JSP. We can make it avialble by using page directive as follows..

<%@ page isErrorPage="true" %>

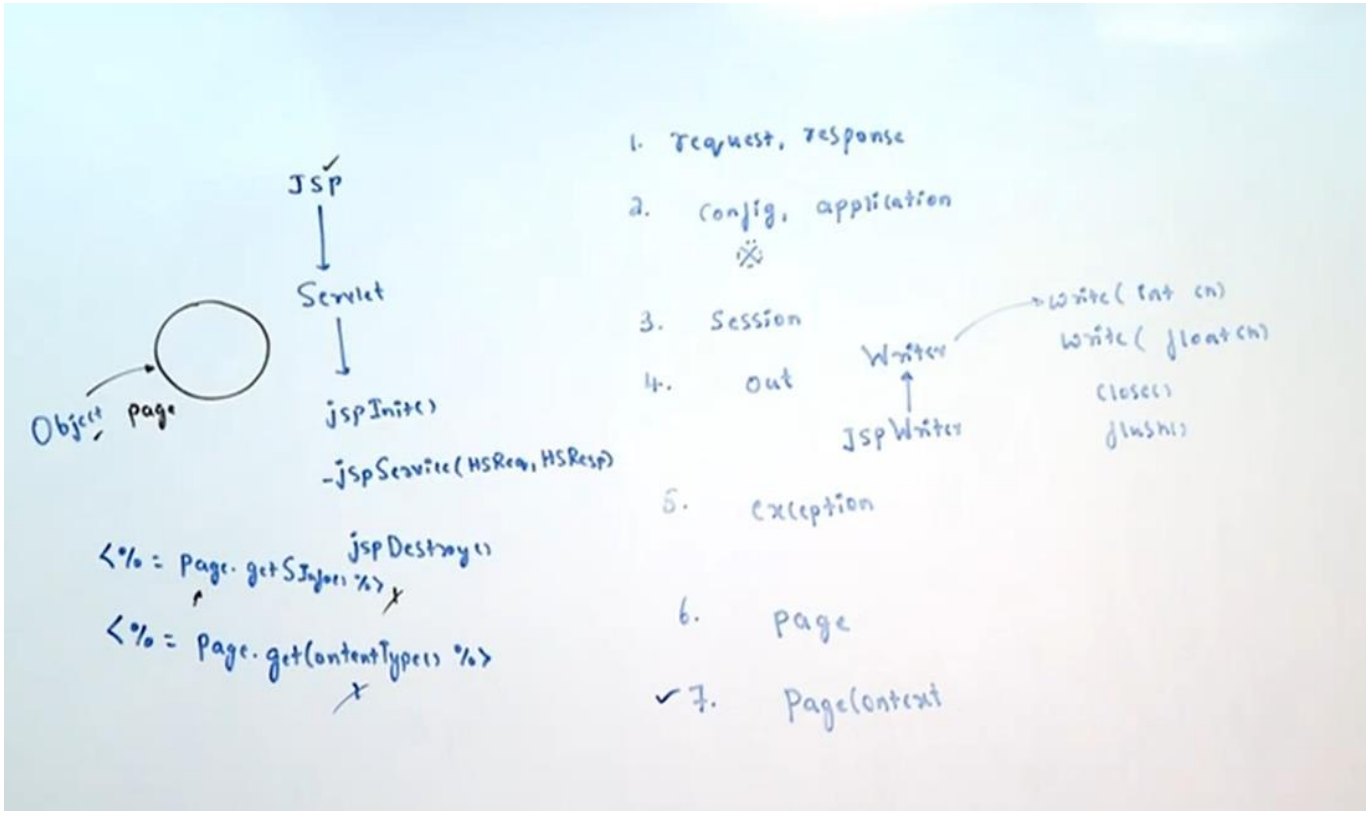
JspDay5:-

JSP(JAVA SERVER PAGES)
==========
Note:
JspPage
1. jspInit()
2. jspDestroy()

HttpJspPage
1. _jspService(request,response)

JSP Implicit Objects
==========
The following is the list of all possible 9 JSP implicit objects.
1) request ➔ HttpServletRequest(I)
2) response ➔ HttpServletResponse(I)
3) config ➔ ServletConfig(I)
4) application ➔ ServletContext(I)
5) session ➔ HttpSession(I)
6) out ➔ javax.serlvet.jsp.JspWriter(AC)
7) page ➔ java.lang.Object(CC)
8) pageContext ➔ javax.servlet.jsp.PageContext(AC)
9) exception ➔ java.lang.Throwable(CC)

pageContext implicit object
==========



The pageContext implicit object is of type javax.servlet.jsp.PageContext.
It is abstract class and web server vendor is responsible to provide implementation.
PageContext is the child class of JspContext.

JspContext ---------> Simple Custom Tag (JSP 2.0 V)
|
|
PageContext ---------> Classic Custom Tag (JSP 1.1 V)

We can use pageContext implicit object for the following 3 purposes.

1) To get all other implicit objects
2) To Perform Request Dispatching activities(include and forward)
3) To perform attribute management in any scopes(dynamic data management)

a) Getting JSP Implicit objects from pageContext:

By using pageContext implicit object,we can get all other JSP implicit objects. i.e pageContext acts as single point of contact for all other implicit objects.

PageContext class contains the following methods for this.

1) request ➔ getRequest()
2) response ➔ getResponse()
3) config ➔ getServletConfig()
4) application ➔ getServletContext()
5) session ➔ getSession()
6) out ➔ getOut()
7) page ➔ getPage()
8) exception ➔ getException()

These methods are not useful within the JSP. We can use these methods outside of JSP, mostly in Custom Tag Handlers.

b) Request Dispatching by using pageContext:

We can perform request dispatching by using the following methods of PageContext.

1) public void forward(String target)
2) public void include(String target)

The target resource can be specified by either relative path or absolute path.

Eg:

first.jsp

```
=========
<h1>Hello This is First JSP</h1>
  <%
       pageContext.forward("second.jsp");
  %>
```

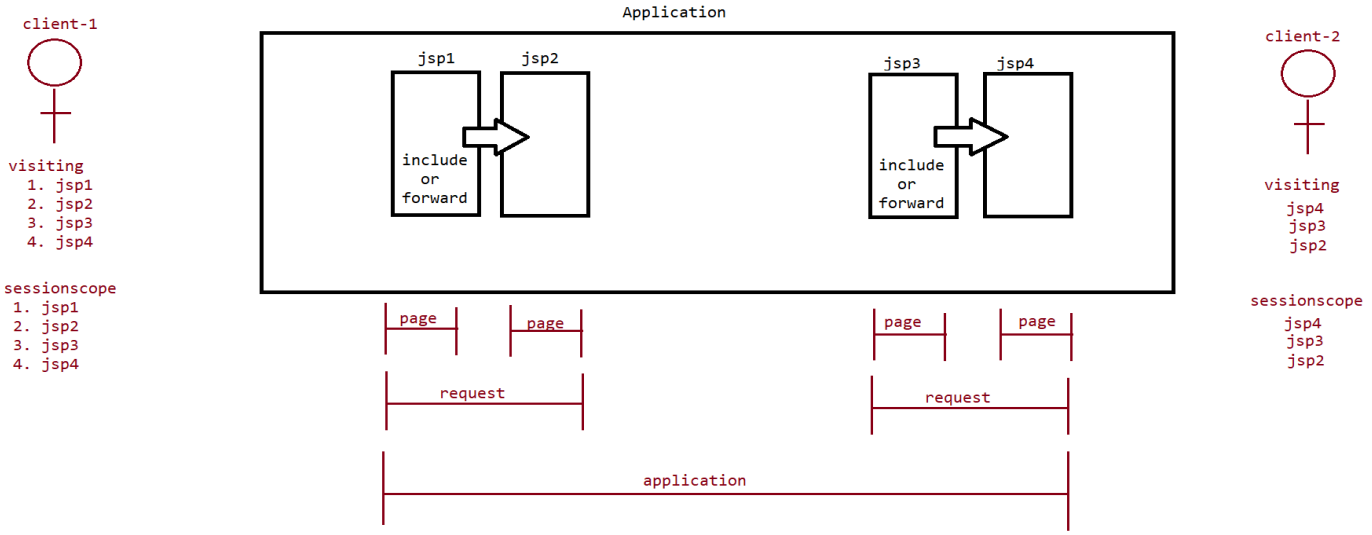
second.jsp

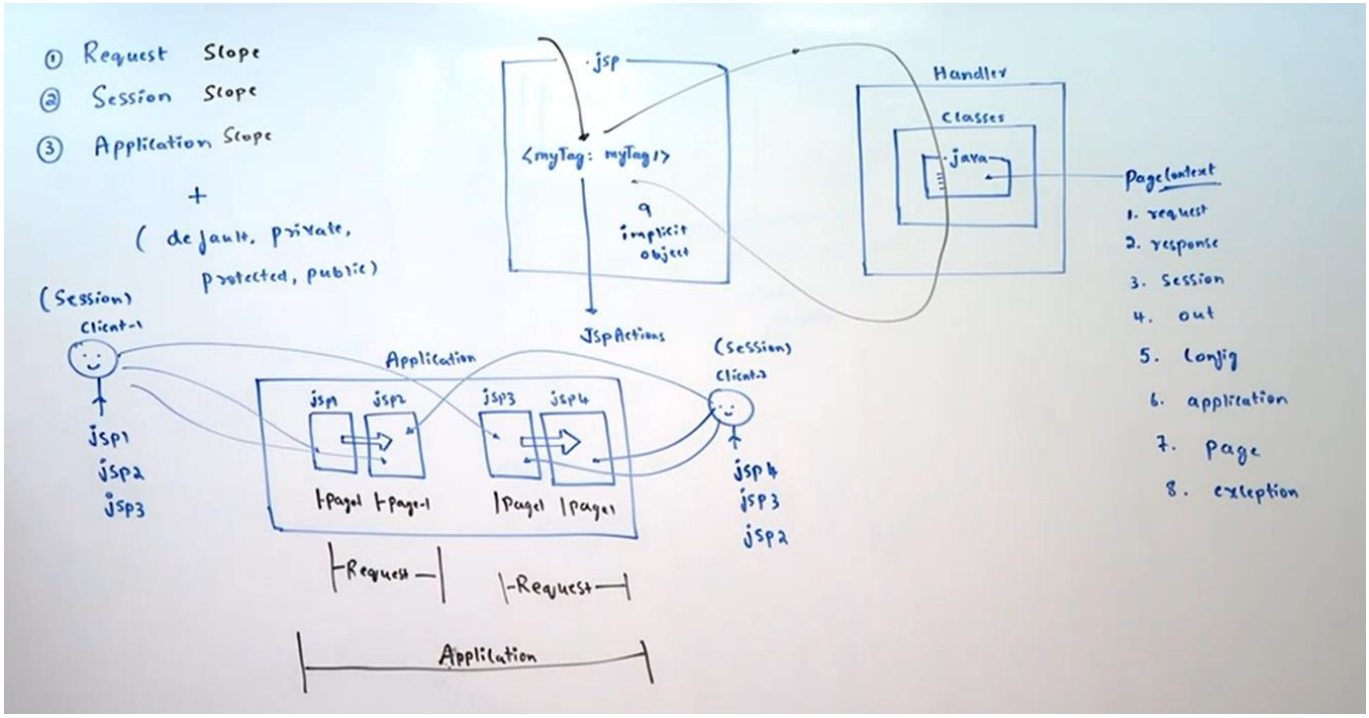
```
==========
<h1>Hello This is Second JSP</h1>
```

forward:
o/p: Hello This is Second JSP

include:
o/p:
Hello This is First JSP
Hello This is Second JSP

c) Attribute Management in any scope:
By using pageContext implicit object we can perform attribute management in any scope.(Will be covered in detail in the next topic: JSP Scopes)



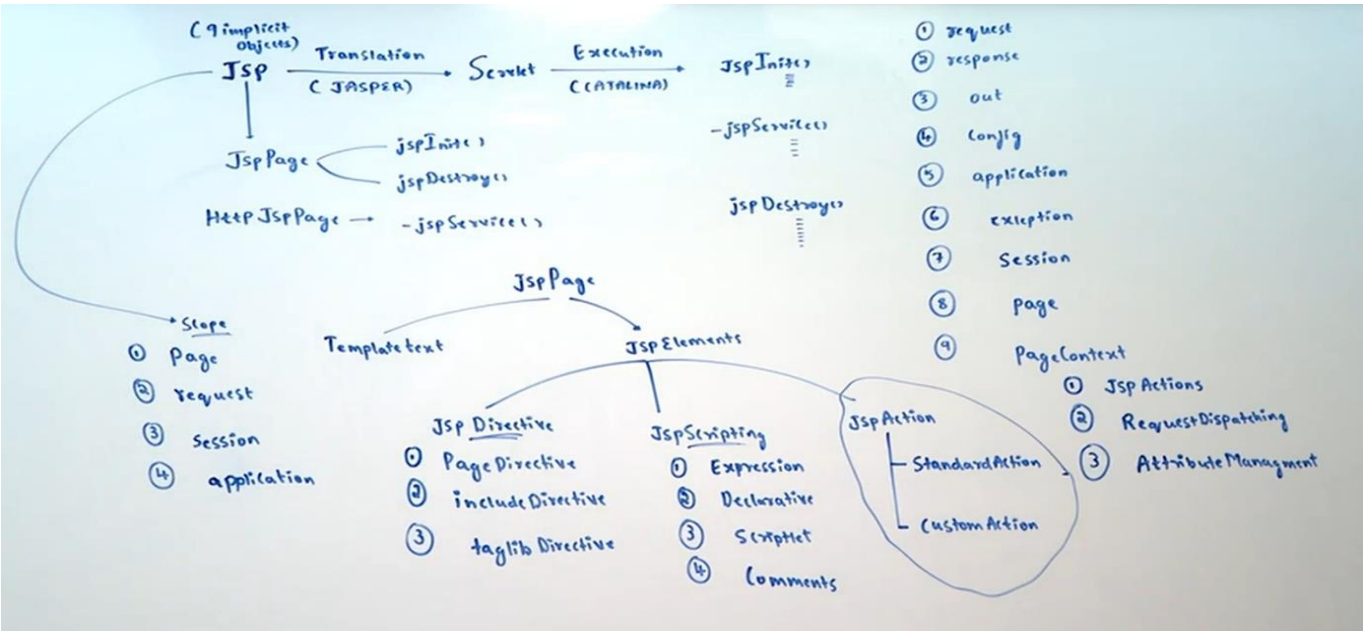


JspDay6:-

JSP(JAVA SERVER PAGES)
==========
Note:
JspPage
1. jspInit()
2. jspDestroy()

HttpJspPage
1. _jspService(request,response)

JSP Implicit Objects
==========
The following is the list of all possible 9 JSP implicit objects.
1) request ➔ HttpServletRequest(I)
2) response ➔ HttpServletResponse(I)
3) config ➔ ServletConfig(I)
4) application ➔ ServletContext(I)
5) session ➔ HttpSession(I)
6) out ➔ javax.serlvet.jsp.JspWriter(AC)
7) page ➔ java.lang.Object(CC)
8) pageContext ➔ javax.servlet.jsp.PageContext(AC)
9) exception ➔ java.lang.Throwable(CC)



JSPActions

==========

Objective:

For the given design problem write code by using the following standard actions

1.<jsp:useBean> with attributes id, type, class and scope

2.<jsp:getProperty>

3.<jsp:setProperty>

We are generally using scripting elements in the JSP. This approach is very easy approach at the beginners level.

test.jsp:

```
<%!
        public int squareIt(int x) {
                return x*x;
        }
 %>
 <h1> The Square of 4 is :<%= squareIt(4) %></br>
      The Square of 5 is :<%= squareIt(5) %></br>
 </h1>
```

This approach has several serious disadvantages…

1. There is no clear separation of presentation and business logic. The person who is writing this JSP should have compulsory the knowledge of both java and html, which may not possible always.

2. This approach does not promote reusability.

3. It reduces readability of the code also.

We can resolve these problems by encapsulating total business logic inside Java Bean.

CalculatorBean.java

==========

```
package com.abc.bean;
public class CalculatorBean {

        public int squareInt(int i) {
                return i * i;
        }

}
```

index.jsp

========

```
<jsp:useBean id="calculator" class="com.abc.bean.CalculatorBean" />
<h1 style='color:red; text-align:center;'>
        The square  of 10 is <%= calculator.squareInt(10) %><br/><br/>
        The square  of 20 is <%= calculator.squareInt(20) %>
</h1>
```

http://localhost:9999/Jsp1/index.jsp

The advantages of this approach are:

1. Separation of Presentation and Business Logic

Presentation logic is available in the JSP, where as business Logic is available in the java class, so that readability of the code will be improved.

2. Separation of Responsibilities

Java Developer can concentrate on business logic where as HTML Page Designer can concentrate on Presentation Logic. As both can work simultaneously we can reduce project development time.

3. It promotes reusability of the code

where ever squareIt() functionality is required we can use same bean without rewriting.

Eg: We can purchase a bean for File uploading and we can start uploading of files within minutes...

Java Bean

=========

Java Bean is a simple java class.

To use bean inside JSP,the bean class has to follow the following rules.

1. The class should contain public no-arg constructor. Otherwise <jsp:useBean> tag won't work. Internally JSP Engine is responsible to create bean object. For this JSP Engine always calls public no-arg constructor.

2. For every Property , Bean class should contain public getter and setter methods.JSP Engine calls these methods to get and set properties of the bean. Otherwise <jsp:getProperty> and <jsp:setProperty> tags won't work.

1. Bean Related Tags : <jsp:useBean>
We can use this tag to make bean object available to the JSP.
There are 2 forms of <jsp:useBean>
1. without body:

```
<jsp:useBean id="calcuator" class="com.abc.Calculator" scope="request" />
```

2. with body:

```
<jsp:useBean id="c" class="pack1.CalculatorBean" >
        body
</jsp:useBean>
```

The main objective of body is to perform initialization for newly created bean object..
If the bean object is already available then <jsp:useBean> won't create any new Object and it will use existing object only. At this time body won't be executed.

```
<jsp:useBean id="calculator" class="com.abc.CalculatorBean"
scope="request" />
```

Attributes of <jsp:useBean>:
<jsp:useBean> can accept the following 5 attributes
1) id
2) class
3) type
4) scope
5) beanName

1. id:
=> This attribute represents the name of the reference variable of the Bean object. By using this id only we can access bean in rest of the JSP.
=> The id attribute is mandatory.
Eg: <jsp:useBean id="person" class="com.abc.Student" type="Person" />
The equivalent java code is : Person person = new Student();

2. class:

This attribute specifies the fully qualified name of java bean class.

=> For this class only JSP Engine will perform instantiation. Hence it should be concrete class and we cannot use abstract class and interfaces.

=> Bean class should compulsory contains public no-arg constructor. Otherwise we will get Instantiation problems(runtime problem).

This attribute is optional & whenever we are not using class attribute compulsory we should use type attribute. In this case <jsp:useBean> won't create any new object and it will always returns existing bean object only.

3. type:

=> This attribute can be used to specify the type of reference variable.

=> The value of type attribute can be concrete class or abstract class or interface.

type attribute is optional, but at that time compulsory we should specify class attribute.

4. scope:

=> This attribute specifies in which scope the JSP Engine has to search for the required bean object.

=> In that scope if the bean object is not already available then JSP engine will create a new bean object & stores that object in the specified scope for the future purpose.

=> The allowed values for the scope attribute are: page,request,session and application.

=> The scope attribute is optional & default value is page scope.

Eg: <jsp:useBean id="calculator" class="com.abc.bean.Calculator" scope="request" />

The equivalent java code is:

```
Calculator calculator=null;
calculator=(Calculator)pageContext.getAttribute("calculator",2);
if(calculator==null) {
        calculator = new CalculatorBean();
        pageContext.setAttribute("caclulaotor",calculator,2);
}
//now bean object is available
```

To use session scope compulsory session object should be available in the JSP. otherwise we will get error.

Eg:

```
<%@ page session="false" %>
<jsp:useBean id="calculator" class="com.abc.bean.Calculator" scope="session" />
```

Error: Illegal for useBean to use session scope when JSP page declares (via page directive) that it does not participate in sessions

5. beanName:

There may be a chance of using Serialized bean from local file system. In this case we have to use beanName attribute.

Various possible combinations

==========

1. id attribute is mandatory
2. scope attribute is optional and default scope is page scope
3. The attributes class, type, beanName can be used in the following combinations:
   a) class
   b) type
   c) class , type
   d) type, beanName (beanName always associated with type but not class)
4. If class attribute is used, whether we are using type attribute or not, it should be concrete class & should contain public no-arg constructor.

   Eg:

   ```
   <jsp:useBean id="c" class="java.lang.Runnable" />
   ```

   Error:The value for the useBean class attribute java.lang.Runnable is invalid.

5. If we are using only type attribute without class attribute, compulsory bean object should be available in the specified scope. Otherwise we will get InstantiationException.

   Eg:

   ```
   <jsp:useBean id="c" type="java.lang.Runnable" />
   ```

   In this case compulsory Runnable Type object should be available in page scope otherwise we will get: java.lang.InstantiationException: bean c not found within scope.

Consider the class

```
package pack1;
public class CustomerBean{
    //code
}
```

Assume that no CustomerBean object is already created. Which of the following standard action creates a new instance of this and store in the request scope.

1. <jsp:useBean name="c" type="pack1.CustomerBean" /> //invalid
2. <jsp:makeBean name="c" type="pack1.CustomerBean" /> //invalid
3. <jsp:useBean id="c" class="pack1.CustomerBean" scope="request" /> //valid
4. <jsp:useBean id="c" type="pack1.CustomerBean" scope="request" /> //invalid

2. Bean Related Tags : <jsp: getProperty >
We can use this standard action to get properties of the bean object.
Eg:
<%
CustomerBean c = new CustomerBean();
out.println(c.getName());==>getting and printing
%>
equivalent code is
<jsp:useBean id="c" class="CustomerBean" />
<jsp:getProperty name="c" property="name" />

<jsp:getProperty> contains the following 2 attributes
1. name:
The name of bean instance from which required property has to get.
It is exactly equal to id attribute of <jsp:useBean>
2. property:
The name of java bean property which has to retrieve.

Both attributes are mandatory.
Note:
<jsp:getProperty> tag internally calls getter method. Hence bean class should compulsory contains corresponding getter method. otherwise <jsp:getProperty> tag won't work.

Demo Program for <jsp:useBean> and <jsp:getProperty>:

index.jsp
=========
```
<jsp:useBean id="employee" class='com.abc.bean.Employee'>
        <h1>
                EID
                <jsp:getProperty name='employee' property='eid' />
```

```
	</h1><br/>
	<h1>
		ENAME
		<jsp:getProperty name='employee' property='ename' />
	</h1><br/>
	<h1>
		ESALARY
		<jsp:getProperty name='employee' property='esalary' />
	</h1><br/>
	<h1>
		EADDR
		<jsp:getProperty name='employee' property='eaddress' />
	</h1>

</jsp:useBean>
```

Employee.java

==========

```
package com.abc.bean;
import java.io.Serializable;
public class Employee implements Serializable {

	private static final long serialVersionUID = 1L;

	private Integer eid=10;
	private String ename="sachin";
	private Integer esalary=10000;
	private String eaddress="MI";

	public Employee() {
		System.out.println("Employee Object created");
	}

	public Integer getEid() {
		return eid;
	}

	public void setEid(Integer eid) {
		this.eid = eid;
	}
```

```
	public String getEname() {
		return ename;
	}

	public void setEname(String ename) {
		this.ename = ename;
	}

	public Integer getEsalary() {
		return esalary;
	}

	public void setEsalary(Integer esalary) {
		this.esalary = esalary;
	}

	public String getEaddress() {
		return eaddress;
	}

	public void setEaddress(String eaddress) {
		this.eaddress = eaddress;
	}
}
```

request => http://localhost:9999/Jsp/index.jsp

3. Bean Related Tags : <jsp: setProperty >

=> This standard action can be used to set properties of bean object.

=> We can use <jsp:setProperty> in the following forms:

```
	<jsp:useBean id="c" class="CustomerBean" />
1. <jsp:setProperty name="c" property="name" value="sachin"/>
                        |
                   c.setName("sachin");
2. <jsp:setProperty name="c" property="name" param="uname"/>
                        |
                   c.setName(request.getParameter("uname"));
```

=> It retrieves the value of specified request parameter and assign its value to the specified bean property.

3. If the request parameter name matches with bean property name, then no need to use param attribute.

```
<jsp:setProperty name="c" property="name" />
                    |
          c.setName(request.getParameter("uname"));
```

4. <jsp:setProperty name="c" property="*" />
   * specifies all properties of the bean.

It iterates through all request parameters and if any parameter name matched with bean property name then it assigns request parameter value to the bean property.

Attributes of <jsp:setProperty>:

1. name:
   It refers the name of bean object whose property has to set. This is exactly same as id attribute of <jsp:useBean>. It is mandatory attribute.

2. property:
   The name of the java bean property which has to be set. It is mandatory attribute.

3. value:
   It specifies the value which has to set to the java bean property.
   It is optional attribute and never comes together with "param" attribute.

4. param:
   This attribute specifies the name of request parameter whose value has to set to the bean property. It is optional attribute and should not come together with value attribute.

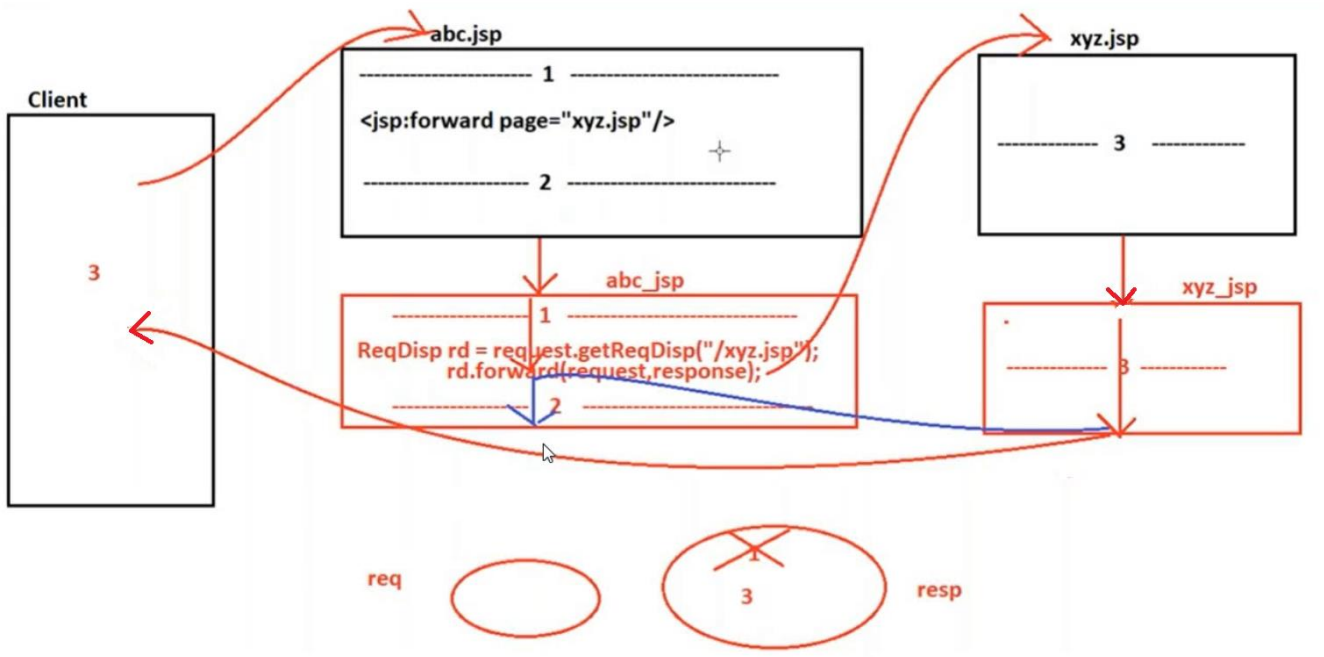
refer: JspApp-06

Note:
If any type of conversions are required then all these conversions will takes care by bean related tags.(String to int)

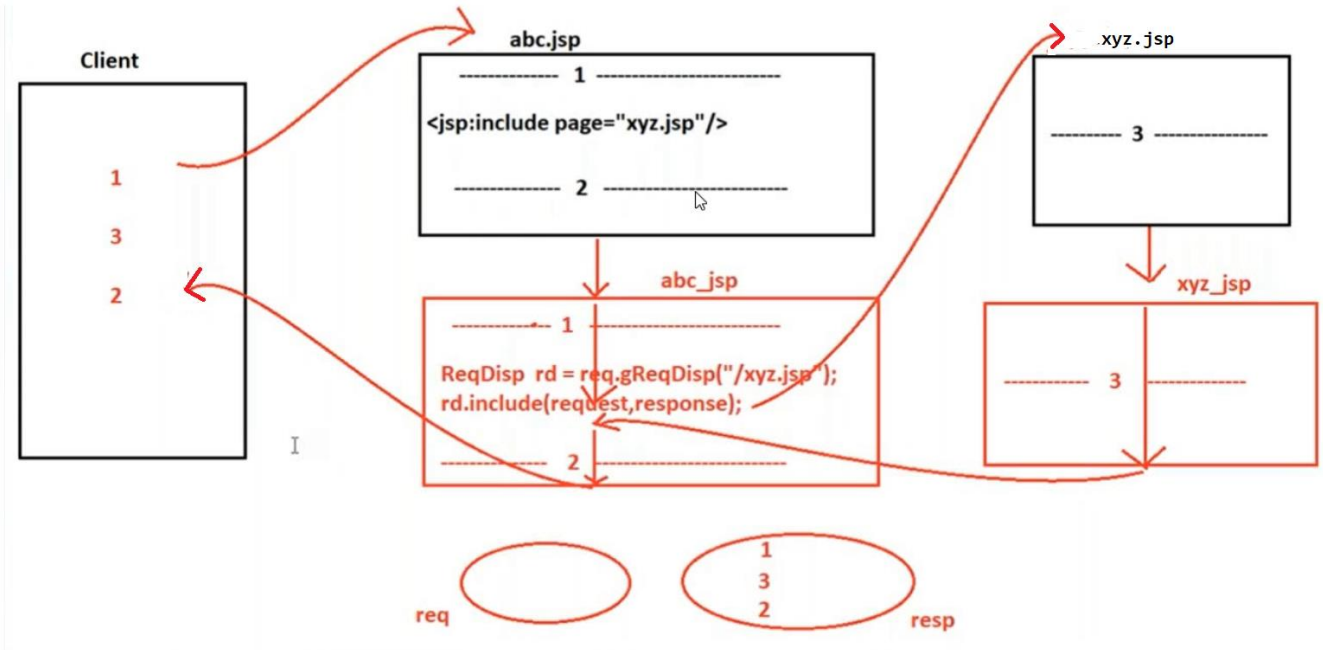
Developing Reusable Web Components

==========

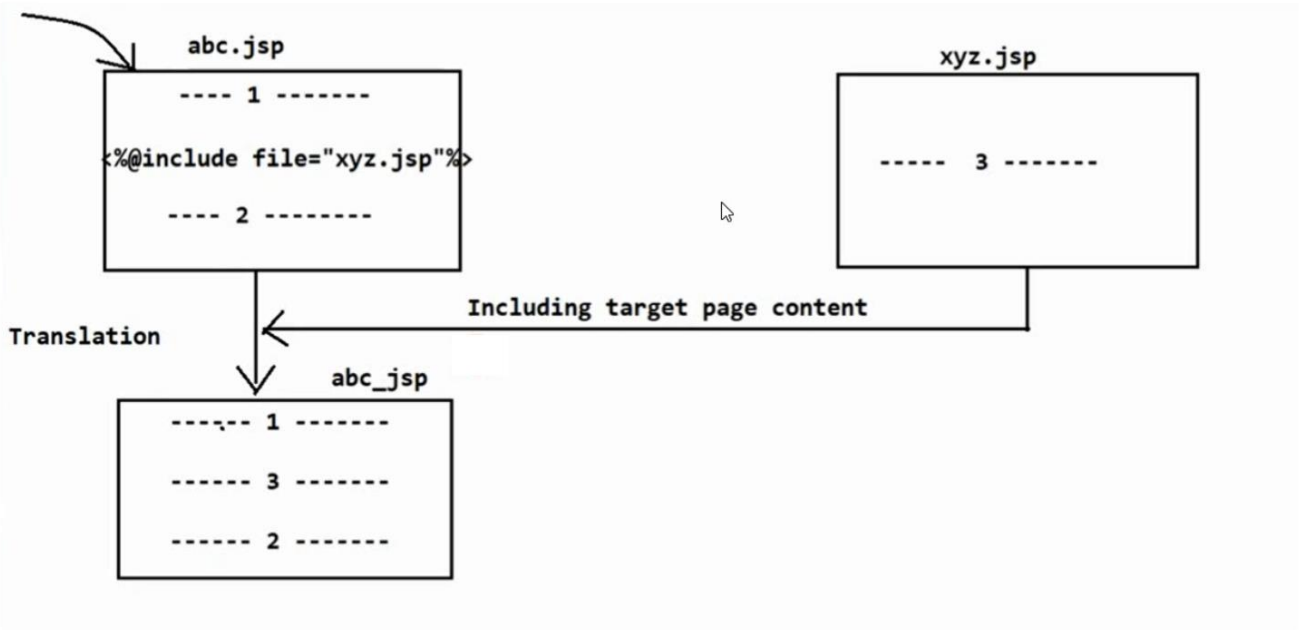
refer: forwardaction.png,



includeaction.png



includedirective.png



We can develop reusable web components by using the following standard actions.
1.<jsp:include>
2.<jsp:forward>
3.<jsp:param>

1.<jsp:include>:
=> The response of header.jsp will be included in the current page response at request processing time. Hence it is dynamic include.
=> This is the tag representation of pageContext.include()
=> This standard action contains the following 2 attributes.
<jsp:include page="header.jsp" flush="true"/>
1. page:
represents the name of the included page. It is mandatory attribute.
2. flush:
It specifies whether the response will be flushed before inclusion or not.
It is optional attribute and default value is false.

Demo Program

==========

header.jsp:

```
<h1>Welcome to ABC...</h1>
```

footer.jsp:

```
<h1>#25/1,1st main road,4th cross Deepanjalinagar B'lore-560026</h1>
```

main.jsp

```
  <jsp:include page="header.jsp" />
       Offered courses are :Java,.Net,Testing...
  <jsp:include page="footer.jsp"/>
```

Note:

In JSPs,we can perform include by using the following 4 ways...

1.include directive:

```
<%@ include file="header.jsp" %>//static inclusion
```

2.include action:

```
<jsp:include page="header.jsp" />//dynamic inclusion
```

3.By using pageContext implicit object:

```
<%
      pageContext.include("header.jsp");//dynamic inclusion
%>
```

4.By using RequestDispatcher:

```
<%
      RequestDispatcher rd = request.getRequestDispatcher("header.jsp");
      rd.include(request,response);
%>
```

2. <jsp:forward>:

=> If the first JSP is responsible for some preliminary processing & Second JSP is responsible for providing complete response, then we should go for forward mechanism.

Syntax:

```
<jsp:forward page="pagename" />
```

first.jsp:

```
<h1>This is First JSP</h1>
<jsp:forward page="second.jsp" />
```

second.jsp:

```
<h1>This is Second JSP</h1>
```

Note: In JSPs we can implement forward mechanism in the following 3 ways.

1. forward action:

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

2. By using pageContext implicit object:

```
<%
    pageContext.forward("second.jsp");
%>
```

3. By using RequestDispatcher:

```
<%
    RequestDispatcher rd = request.getRequestDispatcher("second.jsp");
    rd.forward(request,response);
%>
```

Differences b/w Include Directive and Include Action

Include Directive

1) <%@ include file = "second.jsp" %>
   Contains only one Attribute File.
2) The Content of Target JSP will be included at Translation Time.
   Hence it is also considered as Static Include.
3) For both including and included JSPs, a Single Servlet will be generated. Hence Code sharing between the Components is possible.
4) Relatively Performance is High.
5) There is no Guarantee for Inclusion oflatest Version included JSP. It is Vendor Dependent.
6) If the Target Resource won't change frequently then it is recommended to use Static Include.

Include Action

1) <jsp:include page = "second.jsp" flush = "true"/>
   Contains 2 Attributes Page and File.
2) The Response Target JSP will be included at Runtime. Hence it is also considered as Dynamic Include.
3) For both including and included JSPs, separate Servlets will be generated. Hence Code sharing between the Components is not possible.
4) Relatively Performance is Low.
5) Always latest Version of included Page will be included.
6) If the Target Resource will change frequently then it is recommended to use Dynamic Include

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.

<jsp:include>

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.

SYNTAX: <jsp:include page="" flush=""/>

<jsp:forward>

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.

3. <jsp:param>

while forwarding or including, we are allowed to send parameters to the target JSP.For this we have to use <jsp:param> tag.

```
<jsp:param name="c1" value="JAVA" />
```

<jsp:param> contains the following 2 mandatory attributes

1. name:

   It represents the name of the parameter

2. value:

   It represents the value of the parameter

The parameters which are sending by using <jsp:param> tag are available as form parameters in the target jsp.

first.jsp:

```
<h1>Welcome to ABC...</h1>
  <jsp:include page="second.jsp">
      <jsp:param name="c1" value="JAVA" />
      <jsp:param name="c2" value="TESTING" />
  </jsp:include>
```

second.jsp:

```
<h1>The offered courses are :
  <%= request.getParameter("c1") %> and
  <%= request.getParameter("c2") %>
</h1>
```

Conclusions

==========

Case-1:

<jsp:forward page="http://localhost:7777/jsp2/second.jsp" /> //invalid

<jsp:include page="http://localhost:7777/jsp2/second.jsp" /> //invalid

<%@ include file="http://localhost:7777/jsp2/second.jsp" %> //invalid

The value of the file & page attributes should be relative paths only. We are not allowed to use protocol, server port etc..otherwise we will get 404 status code.

Case-2:

<jsp:forward page="/test" /> //valid

<jsp:include page="/test" /> //valid

<%@ include file="/test" %> //invalid

In the case of forward and include actions, the attribute page can pointing to a servlet.

But in case of include directive, file attribute cannot pointing to a servlet. It can point to html, jsp etc..

Case-3:

<jsp:forward page="second.jsp?c1=java&c2=Tesing" /> //valid

<jsp:include page="second.jsp?c1=java&c2=Tesing" /> //valid

<%@ include file="second.jsp?c1=java&c2=Tesing" %> //invalid

In the case of forward and include actions, we are allowed to pass query string.

But in the case of include directive, we are not allowed to pass query string.

Not Supported by Browser(output will be displayed in the Applets(deprecated))

1. Unused Standard Actions : <jsp:plugin>

We can use <jsp:plugin> to plugin applet in the JSP.

2. Unused Standard Actions : <jsp:fallback>

We can use <jsp:fallback> inside <jsp:plugin> to print information if browser won't provide support for applet.

3. Unused Standard Actions : <jsp:params>

We can use <jsp:params> inside <jsp:plugin> to send parameters from the JSP to the applet.

refer: jspactionsummary.png

JsApp-07

Summary of JSP Standard Actions

Standard Action	Description	Attributes
1) <jsp:useBean>	To Make Bean Object available to the JSP	id, class, type, scope, beanName
2) <jsp:getProperty>	To get and Print Properties of Bean	name, property
3) <jsp:setProperty>	To set the Properties of Bean	name, property, value, param
4) <jsp:include>	To include the Response of Target JSP at Request processing Time	page, flush
5) <jsp:forward>	To forward a Request from one JSP to another JSP	page
6) <jsp:param>	To send Parameters to the Target JSP while performing forward and include	name, value
7) <jsp:plugin>	To plug in Applet in our JSP	type, code, codeBase.....
8) <jsp:fallback>	To display some Message if Browser won't provide Support for Applets	N/A
9) <jsp:params>	To send Parameters from JSP to Applet	N/A

CustomActions

==========

=> Standard actions, EL and JSTL are not succeeded to separate java code completely from the JSP.

For example our requirement is to provide sports information or movie information etc.

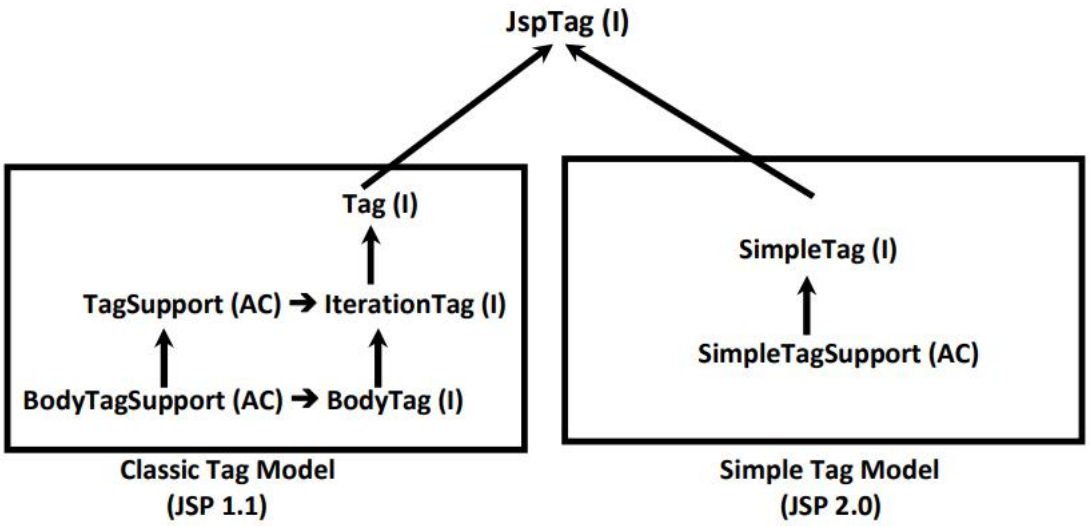
There are no standard actions defined for these requirement.

=> We can define our own tags to meet our requirement from jsp 1.1 version onwards and these tags are nothing but custom tags.

=> All custom tags are divided into 3 categories.

1. Classic Tag Model(JSP 1.1V)
2. Simple Tag Model(JSP 2.0V)
3. Tag Files(JSP 2.0V)

refer: customactions.png



Components of Custom Tag Application

==========

Custom tag application contains the following 3 components

1.Tag Handler class:

=> It is a simple java class which defines entire required functionality

=> Every tag handler class should compulsory implements Tag interface either directly or indirectly.

=> Web container is responsible for creation of Tag Handler object. For this it always invokes public no-arg constructor. Hence every tag handler class should compulsory contains public no-arg constructor.

2.TLD file(Tag Library Descriptor):

=> It is an xml file which provides mapping b/w JSP(where custom tag functionality is required) and tag handler class(where custom tag functionality is available).

```
<taglib version="2.1">
  <tlib-version>1.2</tlib-version>
  <tag>
      <name>tagName</name>
      <tag-class>fully Qualifiedname of the class</tag-class>
  </tag>
</taglib>
```

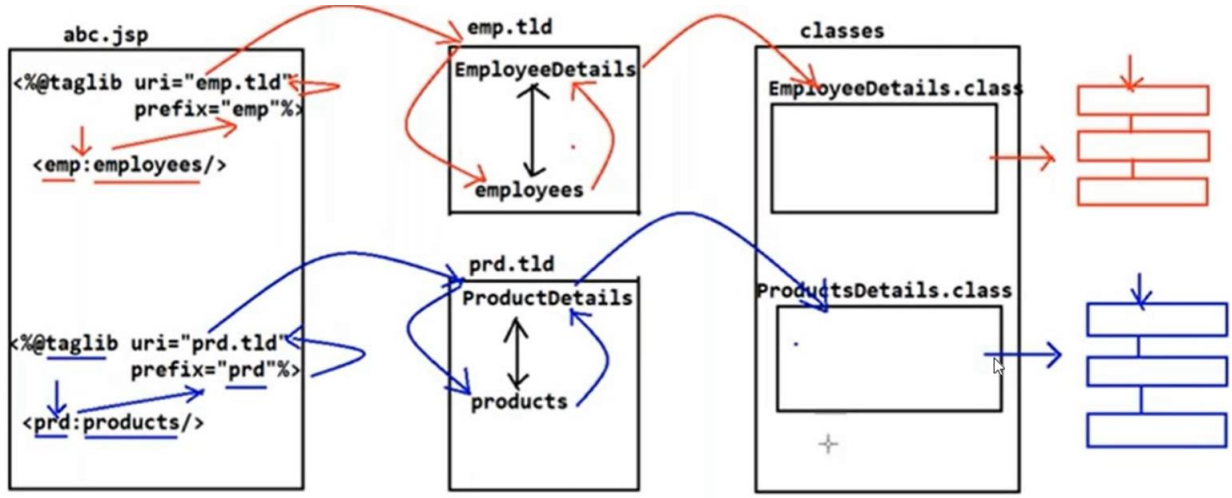
3.taglib directive:

=> It makes custom tag functionality available to the jsp. It defines the location of the tld file.

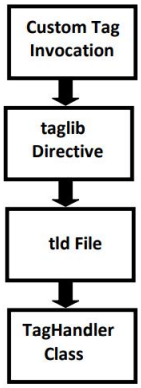
Execution Flow of custom tag application

==========

refer:executionactions.png



Execution Flow of Custom tag application



1. Whenever jsp engine encounters a custom tag, it identifies prefix and checks for the corresponding taglib directive with matched prefix.

2. From the taglib directive jsp engine identifies the location of the tld file
3. From the tld file JSP engine identifies the corresponding Tag handler class.
4. JSP engine executes tag handler class and provides required functionality to the JSP.

Tag Extension API:

We can build custom tags by using only one package:

javax.servlet.jsp.tagext

This package contains the following interfaces and classes to build custom tags.

1. Tag(I):

It is the base interface for all custom tag handlers.

Every tag handler class should compulsory implement this interface either directly or indirectly.

This interface defines 6 methods which are applicable for any Tag Handler object

We should go for this interface if we are not manipulating Tag body and iteration is not required.

Tag interface defines the following 6 methods:

1. setPageContext()
2. setParent()
3. doStartTag()
4. doEndTag()
5. release()
6. getParent()

Tag interface defines the following 4 constants

EVAL_BODY_INCLUDE

SKIP_BODY

EVAL_PAGE

SKIP_PAGE

2. IterationTag(I):

It is the child interface of Tag. We should go for this interface ,if we want to consider tag body multiple times without any manipulation.

It contains only one extra method: doAfterBody()

3. BodyTag(I):

It is the child interface of IterationTag. If we want to manipulate Tag body then we should go for BodyTag.

This interface defines 2 extra methods

setBodyContent() and doInitBody()

4. TagSupport(C):

It implements IterationTag interface and provides default implementation for all its methods.

It acts as base class to develop simple and Iteration tags.
More or less this class acts as adapter class for Tag and IterationTag interfaces.

5. BodyTagSupport(C):
This class implements BodyTag interface and provides default implementation for all its methods.
It is the child class of TagSupport.
We can use this class as Base class for implementing Customtags that processes tag body.

6.BodyContent(C):
BodyContent object acts as a buffer to hold tag body.
It extends JspWriter.
We have to use BodyContent class only in BodyTag interface and BodyTagSupport class.

Life Cycle of Tag Handler that implements Tag Interface

==========

1.Whenever JSP engine encounters a custom tag in the JSP, it will identify the corresponding Tag Handler class through tag lib directive and tld file.

2.Web container creates an instance of TagHandler by executing public no-arg constructor if it is not already available.

3.JSP Engine calls setPageContext() method to make pageContext object available to Tag Handler class.
public void setPageContext(PageContext p)
By using pageContext implicit object, Tag Handler class can get all other implicit objects and can get attributes from various scopes.

4.JSP Engine calls setParent() method to make Parent Tag object available to Tag Handler. This is helpful in nested tags.
public void setParent(Tag t)

5.Setting Attributes

attributes in custom tags are exactly similar to properties of Java beans. If a custom tag has an attribute then compulsory Tag Handler class should contain instance variable and the corresponding setter method.

JSP engine calls setter methods for each attribute.

6. JSP engine calls doStartTag()

public int doStartTag() throws JspException

We can define entire tag functionality in this method only.

doStartTag() method can return either EVAL_BODY_INCLUDE or SKIP_BODY. If it returns EVAL_BODY_INCLUDE then tag body will be included in the result. If it returns SKIP_BODY then JSP Engine won't consider tag body.

7.JSP Engine calls doEndTag()

public int doEndTag()throws JspException

doEndTag() can return either EVAL_PAGE or SKIP_PAGE.

If it returns EVAL_PAGE then rest of the JSP will be executed normally.

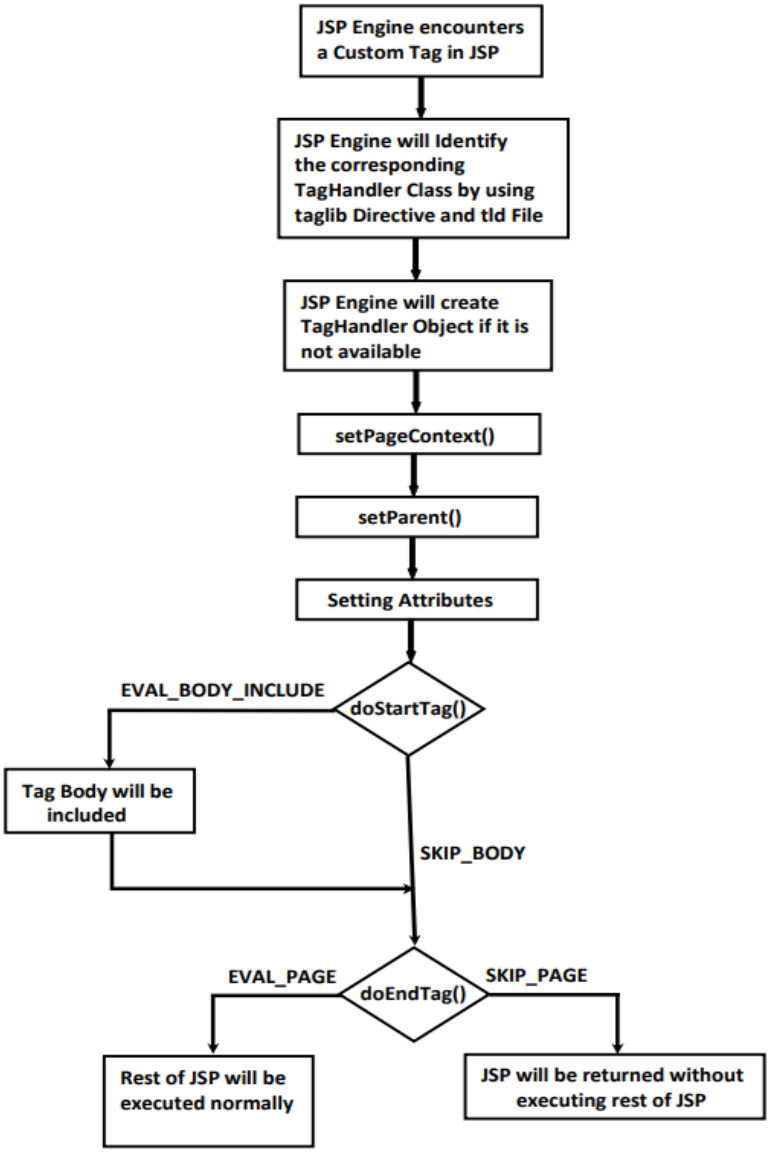
If it returns SKIP_PAGE then JSP page will be returned without executing rest of the JSP.

8. Finally JSP Engine calls release() method to perform cleanup activities whenever tag handler object no longer required.

public void release()

refer: flow chart of taghandlerclass.png

Flowchart for Tag Handler Life Cycle



How to map taglib directive with TLD file

==========

1. By hard coding the location of tld file in the taglib directive

   <%@taglib prefix="mine" uri="/WEB-INF/MyTld.tld" %>

2. Instead of hard coding the location and name of tld file in jsp,we can specify through web.xml also.

Eg: <%@ taglib prefix="mine" uri="www.abcfortech.com" %>

```
<web-app>
  <jsp-config>
    <taglib>
      <taglib-uri>www.abcfortech.com</taglib-uri>
      <taglib-location>/WEB-INF/MyTld.tld</taglib-location>
    </taglib>
  </jsp-config>
</web-app>
```

3. We can map taglib directly to the tld file by using uri attribute.

```
<%@ taglib prefix="mine" uri="www.abcfortech.com" %>
 <taglib version="2.1" >
   <tlib-version>1.2</tlib-version>
   <uri>www.abcfortech.com</uri>
   <tag>
     <name>mytag</name>
     <tag-class>tags.MyCustomTag</tag-class>
   </tag>
</taglib>
```

Q. In how many ways we can map taglib directive to tld file?

Ans. 3 ways

Structure of TLD file:

```
<taglib version="2.1" >
  <tlib-version>1.2</tlib-version>
   <uri>www.abcfortech.com</uri>
   <tag>
        <description>This custom tag for weather report</description>
        <name>mytag</name>
        <tag-class>tags.MyCustomTag</tag-class>
        <body-content>xxx</body-content>
        <attribute>
              <name>
              <required>
              <rtexprevalue>
        </attribute>
   </tag>
</taglib>
```

<body-content>

==========

It describes the type of content allowed in tag body.

The allowed values are:

1.empty:

The body of the tag should be empty. We cannot take any tag body. In this case we can invoke custom tag as follows..

eg1.<mine:mytag>

</mine:mytag>

eg2.<mine:mytag/>

2.tagdependent:

Total tag body will be treated as plain text.

JSP engine sends tag body to the tag handler class without any processing.

3.scriptlet:

Tag body should not contain any scripting elements(i.e. scriptlet, expressions etc),But standard actions and EL expressions are allowed.

4.jsp:

No restrictions on tag body. whatever allowed in JSP is by default allowed in tag body also.

The default value is jsp.

refer: JspApp-08

JspDay7:-

JSP(JAVA SERVER PAGES)

==========

Note:

JspPage

1. jspInit()
2. jspDestroy()

HttpJspPage

1. _jspService(request,response)

JSP Implicit Objects

==========

The following is the list of all possible 9 JSP implicit objects.

1) request ➔ HttpServletRequest(I)
2) response ➔ HttpServletResponse(I)
3) config ➔ ServletConfig(I)
4) application ➔ ServletContext(I)
5) session ➔ HttpSession(I)
6) out ➔ javax.serlvet.jsp.JspWriter(AC)
7) page ➔ java.lang.Object(CC)
8) pageContext ➔ javax.servlet.jsp.PageContext(AC)
9) exception ➔ java.lang.Throwable(CC)

JSTL (JSP Standard Tag Library)

==========

SUN people encapsulates the core functionality, which is common to many web applications in the form of JSTL. Programmer can use this predefined library without writing on his own.

The main objective of EL is removing java code from the JSP. But it fails to replace java code which processes some functionality. We can overcome this problem by using JSTL.

Hence the main objective of JSTL is remove java code from the JSP.

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

Total JSTL Library dividing into 5 sub parts.

1.Core Library:

It defines some standard actions to perform programming general stuff like conditions & loops.

It can also perform JSP fundamental tasks such as setting attributes, writing output & redirecting the request to other pages.

2. SQL Library:

Defines several standard actions which can be used for database operations.

3. Functional Library:
Defines several standard actions which can be used for manipulating String objects.

4. FMT(formating) Library:
Defines several standard actions which can be used for formatting numbers and dates for Internationalization purpose.

5. xml Library:
It defines several standard actions useful for writing and formatting xml data.

Core Library:
JSTL Core Library divided into the following 4 parts based on functionality.

1. General Purpose Tags
<c:out>
<c:set>
<c:remove>
<c:catch>

Summary of General Purpose Tags

Tag	Description	Attribute
1) <c:out>	For writing Expression and Template Text to JSP	Value, default, escapeXml
2) <c:set>	To set Attributes, Bean OR Map Properties	var, value, scope target, property
3) <c:remove>	For removing Attributes	var, scope
4) <c:catch>	For supporting an Expression and to continue rest of Jsp normally	var

2. Conditional Tags
<c:if>
<c:choose>
<c:when>
<c:otherwise>

Summary of Conditional Tags

Tag	Description	Attributes
1) <c:if>	To implement Core Java if - Statement	test, var, scope
2) <c:choose> <c:when> <c:otherwise>	To implement if - else & switch stmt	<c:choose> & <c:otherwise> doesn't contain any Attribute but <c:when> should contain only one Attribute i.e. test

3. Iteration Tags
<c:forEach>
<c:forTokens>

4. URL Related Tags
<c:import>
<c:url>
<c:redirect>
<c:param>

Summary of Iteration Tags

Tag	Description	Attributes
1) <c:foreach>	General Purpose for Loop	items, begin, end, step, var, VarStatus
2) <c:forTokens>	Specialized for String Tokenization	items, delims, begin, end, step, var, VarStatus

Summary of URL related Tags

Tag	Description	Attributes
1) <c:import>	For importing the Response of other Pages at Request processing time.	url, var, scope, varReader, context
2) <c:redirect>	To redirect the Request to other Web Components.	url, context
3) <c:url>	To rewrite URL for appending Session Information & Parameters (Form)	value, var, scope, context
4) <c:param>	To send Parameters while importing & redirecting.	name, value

Installing JSTL

==========

By default JSTL functionality is not available to the JSP. We can provide JSTL Functionality by placing the following jar files in web application's lib folder.

1. jstl.jar:
   Defines several API classes defined by SUN.
2. standard.jar:
   Contains implementation classes provided by vendor.

Note:

It is recommended to place these jar files at server level.(D:\Tomcat 7.0\lib) instead of application level.

To make core library available to JSP, we have to declare taglib directive as follows...

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

General Purpose Tags:

1. <c:out>:
   <c:out>
   where c is prefix and out is name of tag.

This tag can be used for writing Template text data and expressions to the JSP.

form-1:
<c:out value="abc" />
It prints abc to JSP

<c:out value="${param.user}" />
It prints the value of request parameter user to the JSP.

form-2:
If the main value is not available or it evaluates to null, then we can provide default value by using default attribute.
<c:out value="${param.user}" default="Guest" />

test.jsp:
```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<H1>Hello.. <c:out value="${param.uname}" default="Guest" />
```

http://localhost:7777/jsp1/test.jsp?uname=sachin
o/p: Hello sachin

http://localhost:7777/jsp1/test.jsp
o/p: Hello Guest

Note:
<c:out> can accept the following 3 attributes
1.value
2.default
3.escapeXml

2.<c:set>
We can use <c:set> to set attributes in any scope and to set map and bean properties also.

Form-1:
We can use <c:set> to set attributes in any scope
<c:set var="name of the attribute" value="attributevalue" scope="request" />

scope attribute is optional and default scope is page scope.

test.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <c:set var="x" value="10" scope="request" />
  <c:set var="y" value="20" scope="request" />
  <c:set var="sum" value="${x+y}" scope="session" />
<h1>The result is : <c:out value="${sum}" /></h1>
```

form-2:

We can set map or bean properties by using <c:set>

We can specify bean or map object by using target attribute and property name by using property attribute.

```
<c:set target="customer" property="name" value="sachin" />
```

The attributes for the <c:set> are:

1.var 2. value 3. scope 4. target 5. property

3. <c:remove>:

To remove attributes in the specified scope we can use this tag.

This tag can take the following 2 attributes

1.var ➔ Name of the Attribute

2.scope ➔ The Scope in which Attribute Present

Eg: <c:remove var="x" scope="session" />

If the scope is not specified then the container will search in the page scope for the required attribute. If the attribute is not available then it will search in the request scope followed by session and application scopes.

test.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <c:set var="x" value="40" scope="page" />
  <c:set var="y" value="60" scope="page" />
  <c:set var="sum" value="${x+y}" scope="session" />
 <h1> The result is : <c:out value="${sum}" /><br>
  <c:remove var="x" />
  <c:remove var="y" />
  <c:remove var="sum" />
  The result is : <c:out value="${sum}" default="1000"/><br>
  </h1>
```

2.

<c:catch>:
This tag can be used to catch an exception within the JSP instead of forwarding to the error page.The risky code we have to place as the body of <c:catch>

```
<c:catch >
  Risky Code
</c:catch>
```

If an exception raised in the risky code, then this tag suppresses that exception and rest of the JSP will be executed normally.
If an exception is raised we can hold that by using var attribute, which is the page scoped attribute.

test.jsp:

```
 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <h1>
   User Name: ${param.uname}<br>
  <c:catch var ="e" >
    <%
      int age = Integer.parseInt(request.getParameter("uage"));
    %>
    Age:${param.uage}<br>
  </c:catch>

   <c:if test="${ e != null}">
     Oops ...Exception raised : ${e}<br>
   </c:if>
          User Height: ${param.uheight}
  </h1>
```

http://localhost:7777/jsp1/catch1.jsp?uname=pavan&uage=47&uheight=5.11
User Name: pavan
Age:47
User Height: 5.11
http://localhost:7777/jsp1/catch1.jsp?uname=pavan&uage=ten&uheight=5.11
User Name: pavan
Oops ...Exception raised : java.lang.NumberFormatException: For input string: "ten"
User Height: 5.11

Conditional Tags

1.<c:if>

We can use to implement core java if statement.

There are 2 forms are available for <c:if> without body:

<c:if test="test_condition" var="x" scope="session"/>

In this case test_condition will be evaluated & store its results into variable x. In the rest of the page where ever this test condition is required, we can use directly its value without evaluating once again.

In this case both test and var attributes are mandatory. scope attribute is optional and default scope is page.

test.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
        <c:set var="x" value="10" scope="request"/>
        <c:if test="${x ne '10'}" var="y" scope="session" />
   <h1>
         X value is : ${x}<br>
         Test Result is : ${y}
   </h1>
```

with body:

```
<c:if test="test_condition" var="x" scope="session">
    body
</c:if>
```

If test_condition is true then body will be executed otherwise without executing body rest of the jsp will be continued.

In this case also we can store test result inside var attribute.

Both var and scope attributes are optional but test attribute is mandatory.

test.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:set var="x" value="10" scope="request"/>
        <c:if test="${x eq '10'}" >
                <h1> x value is equal to 10</h1>
        </c:if>
```

3.

<c:choose>,<c:when> and <c:otherwise>:

We can use these tags for implementing if-else and switch statements.

1. Implementing if-else:

JSTL does not contain any tag for else. We can implement if-else statement by using above tags.

```
<c:choose>
        <c:when test="test_condition">
                Action-1
        </c:when>
        <c:otherwise>
                Action-2
        </c:otherwise>
</c:choose>
```

If test condition is true,Action-1 will be executed otherwise Action-2 will be executed.

Implementing switch statement:

```
 <c:choose>
        <c:when test="test_condition1">
                Action-1
        </c:when>
        <c:when test="test_condition2">
                Action-2
        </c:when>
                .....
        <c:when test="test_conditionN">
                Action-N
        </c:when>
        <c:otherwise>
                Default Action
        </c:otherwise>
 </c:choose>
```

1. <c:choose> should compulsory contains at least one <c:when> tag. But <c:otherwise> is optional.
2. Every <c:when> tag implicitly contains break statement. Hence there is no chance of fall through inside switch.
3. We should take <c:otherwise> as last case only.
4. <c:choose> and <c:otherwise> won't take any attributes but <c:when> can take one mandatory attribute test.

switch.jsp:

```
  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
        <h1> Select The Number:
                <form action="/jsp1/switch.jsp">
```

```
                    <select name="combo" >
                            <option value="1">1</option>
                            <option value="2">2</option>
                            <option value="3">3</option>
                            <option value="4">4</option>
                            <option value="5">5</option>
                            <option value="6">6</option>
                            <option value="7">7</option>
                    </select>
                    <input type="submit"/>
                </form>
        <c:set var="s" value="${param.combo}"/>
        Today is : <c:choose>
                    <c:when test="${s == 1}" >Sunday</c:when>
                    <c:when test="${s == 2}" >Monday</c:when>
                    <c:when test="${s == 3}" >Tuesay</c:when>
                    <c:when test="${s == 4}" >Wednesday</c:when>
                    <c:when test="${s == 5}" >Thursday</c:when>
                    <c:otherwise>Select between 1 and 5</c:otherwise>
                </c:choose>
    </h1>
```

Iteration Tags:
There are 2 iteration tags are available in JSTL.

1. <c:forEach> ➔ General purpose for loop
2. <c:forTokens> ➔ Specialized for splitting Strings similar to StringTokenizer.

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

```
  <c:forEach begin="1" end="10" step="1" >
       <h1>Learning JSTL is very easy!!!!</h1>
  </c:forEach>
```

o/p: 10 times

<c:forEach begin="1" end="10" step="2" > 5 times
<c:forEach begin="4" end="0" step="-1" > Invalid b'z step value should not be -ve.
The begin attribute specifies the index where the loop has to start.
end index specifies the index where the loop has to terminate.
This loop internally maintains counter variable which is incremented by step attribute value.
The default value of step attribute is "1" and it is optional attribute.

form-2: <c:forEach> with var attribute:
<c:forEach> internally maintains a counter variable which can be accessed by using var attribute.
This variable is local variable. Hence from outside of for loop we cannot access.

test.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
    <c:forEach begin="1" end="10" step="2" var="count">
        <h1>Learning JSTL is very easy!!!!---${count}</h1>
    </c:forEach>
```

Form-3: <c:forEach> for iterating through Arrays and Collections:

```
<c:forEach items="collection object" var="current object">
    body
</c:forEach>
```

Items attribute should be either Collection object or array object. This action will iterate over each item in the collection until all elements completion.
We can represent current collection object by using var attribute.

Q. Write a JSP to print all elements of array:

test.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%
    String[] s = {"A","B","C","D"};
    pageContext.setAttribute("s",s);
%>
<c:forEach items="${s}" var="obj">
    <h1>The current Object is : ${obj} </h1>
</c:forEach>
```

Q. Write a JSP to print all request headers information:

test.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
    <table border="2">
        <c:forEach items="${header}" var="hdr" >
            <tr> <td>${hdr.key}</td><td>${hdr.value}</td></tr>
        </c:forEach>
    </table >
```

Q. Write a JSP to print all request parameters information:

test.jsp:

```
<table border="2">
    <c:forEach items="${param}" var="p" >
        <tr> <td>${p.key}</td><td>${p.value}</td></tr>
    </c:forEach>
</table >
```

http://localhost:7777/jsp1/forEach2A.jsp?user=sachin&age=48&course=java&mail=sachin@gmail.com

Form-4: <c:forEach> with varStatus attribute:

varStatus attribute describes the status of iteration like current iteration number, is it first iteration or not etc..

This attribute is of type javax.servlet.jsp.core.TagLoopStatus.

This class contains several methods which are useful during iteration.

1. Object getCurrent()
   returns the current item
2. int getIndex()
   returns the current index(counter value)
3. int getCount()
   returns the number of iterations that have already performed.
4. boolean isFirst()
   Returns true if the current Iteration is the First iteration
5. boolean isLast()
   Returns true if the current iteration is the last iteration
6. Integer getBegin()
   returns start index
7. Integer getEnd()
   returns end index
8. Integer getStep()
   returns the incremented value

test.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:forEach items="durga,pavan,ravi,shiva" varStatus="status" >
    <h1>Is it First Iteration: ${status.first}<br>
    The current Object is : ${status.current}<br>
    The number of iterations already completed:${status.count}<br>
    Is it Last Iteration :${status.last}<br><hr></h1>
</c:forEach>
```

<c:forTokens>:
It is a specialized version of forEach to perform StringTokenization based on some delimiter. It acts as StringTokizer. We can store current token by using var attribute.

test.jsp:
```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:forTokens items="sachin.dravid.ganguly.yuvraj" delims="." var="x">
        <h1>Hello:${x}<br>
</c:forTokens>
```

<c:forTokens> can accept the following extra attributes also.
begin: specifies the index at which iteration should start. The index of first token is zero.
end: specifies the index where iteration should be terminated.
step: counter increment value between iterations
varStatus: to specify the status of iteration

test.jsp:
```
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <c:forTokens items="one,two,three,four,five,six" delims="," var="x" begin="2"
       end="5" step="1">
         <h1>Current Token is : ${x}</h1>
  </c:forTokens>
```

Note:
In the case of <c:forTokens>, items attribute should be String only. But in the case of <c:forEach> the items attribute can be Collection, Array, Map OR String. Hence <c:forTokens> is considered as specialized version of <c:forEach>

URL Related Tags
==========

1. <c:import>
We can use this tag for importing the response of other pages in the current page response at request processing time.(i.e. Dynamic Include)

Form-1:

```
<c:import url="second.jsp" />
```

Eg:

first.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<h1>Welcome to ABC!!!!</h1>
<c:import url="second.jsp" />
```

second.jsp:

```
<h1>The FREE videos are available in www.abcvideos.com</h1>
```

Form-2:

We can import response from outside of current application also.
(i.e cross context communication is possible)

```
<c:import url="/second.jsp" context="/webapp2" />
```

url and context should be specified with absolute Path only i.e should starts with "/"

Form-3:

We can store the result of imported page into a variable specified by var attribute. In the rest of the JSP, where ever that result is required we can use directly that variable without importing once again.

test.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:import url="second.jsp" var="x" scope="request" />
${x}<br>
${x}<br>
${x}<br>
${x}<br>
${x}<br>
```

Form-4:

Another way to store the result of <c:import> is to use Reader object. It is alternative to var attribute.

```
<c:import url="second.jsp" varReader="r" scope="page" />
```

Form-5:
While performing import we can send form parameters also to the Target JSP. For this we have to use <c:param> tag. These parameters are available as form parameters in the target JSP.
first.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
        <h1>Welcome to ABC!!!!</h1>
        <c:import url="second.jsp" >
                <c:param name="c1" value="JAVA"/>
                <c:param name="c2" value="PYTHON"/>
        </c:import>
```

second.jsp:

```
<h1> The offered courses are : ${param.c1} and ${param.c2}
```

2.<c:redirect>:
This action can be used to redirect the request to another page. This is exactly similar to sendRedirect() method of ServletResponse.

Form-1:

```
<c:redirect url="second.jsp" />
```

url can be specified either with Relative path or absolute path

first.jsp:

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

second.jsp:

```
<h1>Hello this is Second JSP</h1>
```

If we send the request to first.jsp then second.jsp will provide response through redirection.

Form-2:
We can redirect request to some other web application's resources also.

```
<c:redirect url="/second.jsp" context="/webapp2" />
```

url and context should be specified with Absolute Path only i.e. should starts with "/"

Form-3:
While performing redirection we can pass form parameters also to target resource.
first.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
    <c:redirect url="second.jsp" >
        <c:param name="c1" value="Java"/>
        <c:param name="c2" value="PHP"/>
    </c:redirect>
```

second.jsp:

```
<h1>Welcome to ABC!!!!<hr>
The offered courses are : ${param.c1} and ${param.c2}
```

3.<c:url>
We can use this standard action to rewrite urls to append session information and form parameters.
Form-1:

```
<c:url value="second.jsp" var="x" scope="request"/>
```

The encoded url with sessionid will be stored inside x.

test.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <c:url value="second.jsp" var="x" scope="request"/>
  <h1>The modified url:${x}</h1>
```

Form-2:

```
<c:url value="/second.jsp" context="/webapp2" var="x" scope="request"/>
```

Form-3:

```
  <c:url value="second.jsp" var="x" >
      <c:param name="c1" value="java" />
      <c:param name="c2" value="php" />
  </c:url>
```

first.jsp:

```
  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
      <c:url value="second.jsp" var="x" >
          <c:param name="c1" value="JAVA" />
          <c:param name="c2" value="PHP" />
      </c:url>
      <h1>The modified url is : ${x}<br> </h1>
      <a href="${x}">Click Here to go to Next Page</a>
```

second.jsp

```
<h1>Offered courses are : ${param.c1} and ${param.c2}</h1>
```

Summary of All Core Library Tags:

1. <c:out> To display expression to the JSP
2. <c:set> to set attributes and bean properties
3. <c:remove> to remove attributes from the specified scope
4. <c:catch> To suppress exception and continue rest of the jsp
5. <c:if> To implement core java if statement
6. <c:choose>,<c:when>,<c:otherwise> To implement if-else and switch statements
9. <c:forEach> To implement general purpose for loop
10. <c:forTokens> For string tokenization purpose
11. <c:import> for dynamic include
12. <c:redirect> for redirection
13. <c:param> to send parameters while importing and redirecting
14. <c:url> to append session information to the url for session management purpose.

JSTL SQL Library

It defines several tags for communicating with database.

To make sql library available to JSP, we have to write the following taglib directive.

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

The following are important sql library tags

1. <sql:setDataSource>
   To create DataSource
2. <sql:query>
   To execute select query
3. <sql:update>
   To execute non-select query(insert|delete|update)
4. <sql:param>
   To provide parameter values in the case of PreparedStatement
5. <sql:dateParam>
   To provide date values in the case of PreparedStatement
6. <sql:transaction>
   To implement transactions

Note: Transaction is a group of sql queries which will be executed on the policy "Either ALL or None".

Note: It is not recommended to use JSTL SQL Library in our JSP, b'z JSP meant for presentation logic but not for business logic and database logic.

test.jsp:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>
      <sql:setDataSource var="ds" driver="oracle.jdbc.OracleDriver"
            url="jdbc:oracle:thin:@localhost:1521:XE"
            user="scott" password="tiger"/>
      <sql:query dataSource="${ds}" var="result">
             SELECT * from Employees
      </sql:query>
      <h1>
            <c:forEach items="${result.rows}" var="row" >
                  ${row.eno}--${row.ename}--${row.esal}--${row.eaddr}<br>
            </c:forEach>
      </h1>
```

~~~~~~

```
 <sql:update dataSource="${ds}" var="count">
      insert into employees values(500,'Lasya',5000,'hyd')
 </sql:update>
  <h1>The number of rows inserted:${count}</h1>
```

~~~~~~

```
  <sql:update dataSource="${ds}" var="count">
     delete from employees where esal>=4000
  </sql:update>
   <h1>The number of rows deleted:${count}</h1>
```

~~~~~~

```
  <sql:update dataSource="${ds}" var="count">
      update employees set esal=7777 where ename='sachin'
  </sql:update>
  <h1>The number of rows updated:${count}</h1>
```
~~~~~~

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
  <sql:update dataSource="${ds}" var="count">
     update employees set esal=? where eno=?
        <sql:param value="9999"/>
        <sql:param value="100"/>
  </sql:update>
  <h1>The number of rows updated:${count}</h1>
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
 <sql:transaction dataSource="${ds}">
   <sql:update>
         update employees set esal=esal-1000 where ename='sachin'
   </sql:update>
   <sql:update >
        update employees set  esal=esal+1000 where ename='dhoni'
   </sql:update>
  </sql:transaction>
```

JSTL Functional Library

=======================

It defines several tags for general String manipulation operations.
To use Functional Library,we have to write the following taglib directive in JSP.

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

The following are important tags in this library:
1.<fn:contains()>
Eg2:
split() method will split the given string according to the specified separator (delimiter) join() method will join the given tokens into a single string based on separator.

Eg1:

```
 <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <c:set var="s" value="Hello Learning JSTL is Very Easy"/>
  <h1>
```

```
    Length: ${fn:length(s)}<br>
    In Upper Case: ${fn:toUpperCase(s)}<br>
    In Lower Case: ${fn:toLowerCase(s)}<br>
    Sub String from index 6 to 15: ${fn:substring(s,6,15)}<br>
    Is s contains JSTL: ${fn:contains(s,"JSTL")}<br>
    Is s starts with Hello: ${fn:startsWith(s,"Hello")}<br>
    Is s ends with Easy: ${fn:endsWith(s,"Easy")}<br>
```

~~~~~~

Eg2:

```
  <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <c:set var="data" value="sachin,saurav,dhoni,dravid,yuvi"/>
  <c:set var="s" value="${fn:split(data,',')}"/>
  <h1>
       Result of split method:<br>
       <c:forEach items="${s}" var="s1">
            ${s1}<br>
       </c:forEach>
    <c:set var="result" value="${fn:join(s,'-')}"/>
        Result of Joining: ${result}
```

ExpressionLanguage

==========

Agenda:

1.EL Introduction
2.EL Implicit objects
3.EL Operators
4.EL Functions
5.EL introduced in JSP 2.0V.

EL Introduction:

The main objective of EL is to eliminate java code from the JSP.
In general we can use EL with JSTL and Custom Tags for complete elimination of java code.

Eg1:

To print the value of request parameter "user"
Standard: <%= request.getParameter("user") %>
EL: ${param.user}
~~~~~~

Eg2:
To print the value of x attribute
Standard: <%= pageContext.findAttribute("x") %>
EL: ${x}

Note:
If the specified attribute is not available then standard syntax will print null, but EL syntax will print blank space(i.e. won't print anything)
EL suppresses null
EL syntax:
If we are using any variable in EL syntax, compulsory it should be some attribute in some scope.
${x}

EL Implicit Objects:
EL contains 11 implicit objects. The power of EL is just because of these implicit objects only.
The following is the list of all EL Implicit objects
1) pageScope
2) requestScope
3) sessionScope
4) applicationScope
5) param
6) paramValues
7) header
8) headerValues
9) cookie
10) initParam
11) pageContext

1. pageScope,requestScope,sessionScope and applicationScope
We can use these implicit objects to retrieve attributes of a particular scope

pageScope ➔ pageContext.getAttribute()
requestScope ➔ request.getAttribute()
sessionScope ➔ session.getAttribute()
applicationScope ➔ application.getAttribute()

Eg1:
To get the value of session scoped attribute x
${sessionScope.x}

Eg2:
To get the value of request scoped attribute x
${requestScope.x}

Eg3:
${x}

JSP Engine first checks in page scope for the attribute "x". If it is available then JSP Engine prints its value. If it is not available then JSP engine will search in request scope followed by session scope and application scope. It is exactly same as findAttribute()

test.jsp:

```
<%
        pageContext.setAttribute("p","Pavan");
        pageContext.setAttribute("p","ravi",2);
        pageContext.setAttribute("p","sunny",3);
        pageContext.setAttribute("p","anushka",4);
%>
        <h1>
        ${pageScope.p}<br>
         ${requestScope.p}<br>
        ${sessionScope.p}<br>
        ${applicationScope.p}<br>
        ${p}<br>
        </h1>
```

Output:
Pavan
ravi
sunny
anushka
Pavan

2. param and paramValues
We can use these implicit objects to retrieve request form parameters.
param ➔ getParameter()
paramValues ➔ getParameterValues()
${param.x}
It prints the value of form parameter x
If the parameter associated with multiple values then we will get only first value.

If the specified parameter not available then we will get blank space
${paramValues.x}
Internally it returns String[] which contains all values associated x. as we are printing that String[],
internally toString() method will be called, which prints result in the following form
[Ljava.lang.String;@1270b73
${paramValues.x[0]} ➔ It prints first value of x
${paramValues.x[1]} ➔ It prints second value of x
${paramValues.x[100]} ➔ prints blank space as EL suppresses ArrayIndexOutOfBoundsException

Demo Program:
details.jsp:

```
<form action="el3.jsp" >
 Enter name: <input type=text name=uname><br>
 Enter EmpId:<input type=text name=eid><br>
 Enter Food1:<input type=text name=food><br>
 Enter Food2:<input type=text name=food><br>
 <input type=submit>
</form>
```

el3.jsp:

```
<h1>
 Name:${param.uname}<br>
 Emp Id: ${param.eid}<br>
 Food: ${param.food}<br>
 Food[]:${paramValues.food}<br>
 Food1: ${paramValues.food[0]}<br>
 Food2: ${paramValues.food[1]}<br>
 Food100: ${paramValues.food[100]}<br>
</h1>
```

3. header and headerValues:
These are exactly same as param and paramValues except that these are for retrieving request
headers.
header ➔ request.getHeader()
headerValues ➔ request.getHeaders()
Eg:

```
<h1>${header.accept}<br>
<h1>${header.host}<br>
```

<h1>${headerValues.host[0]}

4. cookie implicit object :
By using this implicit object we can get cookies associated with the request.
cookie ➔ request.getCookies()
<h1>${cookie.JSESSIONID.name}
${cookie.JSESSIONID.value}

5. initParam implicit object:
We can use initParam implicit object to access context parameters declared in web.xml
initParam ➔ context.getInitParameter()

Demo Program:
web.xml:

```
<web-app>
 <context-param>
     <param-name>user</param-name>
     <param-value>scott</param-value>
 </context-param>
 <context-param>
      <param-name>pwd</param-name>
      <param-value>tiger</param-value>
 </context-param>
</web-app>
```

el6.jsp:

```
<h1>User: ${initParam.user}<br>
<h1>Pwd: ${initParam.pwd}<br>
<h1>Mail Id: ${initParam.mailId}<br>
```

Note:
If the specified initParameter is not available then we will get blank space b'z EL suppresses null

6. pageContext implicit object:
This is the only common implicit object between JSP and EL.
This is the only EL implicit object which is a non-map object.
By using this implicit object we can access all other JSP implicit objects in EL.
Eg:
${request.method} ➔ invalid

${pageContext.request.method} ➔ valid
${session.id} ➔ invalid
${pageContext.session.id} ➔ valid
In EL, request and session implicit objects are not available.
Note: El handles null and ArrayIndexOutOfBoundsException very nicely and prints blank space.

EL Operators
==========
EL contains its own specific operators. The following is the list of all possible operators.
1. Property Access operator(.)
2. Collection Access Operator([])
3. Arithmetic Operators
4. Relational Operators
5. Logical Operators

1. Property Access operator(.)
Syntax: ${leftvariable.rightvariable}
Left Variable ➔ Map OR Bean
Right Variable ➔ Map Key OR Property
Should be Valid Java Identifier
Eg:
${customer.name} ➔ valid
${pageContext.request.method} ➔ valid
${initParam.user} ➔ valid
${customer.1234} ➔ invalid

2. Collection access operator([]):
Syntax: ${leftvariable[rightvariable]}
map ➔ key
bean ➔ property
array ➔ index
list ➔ index
key or property should be enclosed in either single quotes or double quotes. For index quotes are
optional.
Case-1:working with Map:
${initParam["user"]}
${initParam['user']}

${initParam[user]} blank space
If we are not keeping quotes EL assumes that the key is an attribute stored in some scope. If that
attribute is not available then we will get blank space as output.

Q. Which of the following is the valid way of retrieving request header "accept" value?

1. ${header.accept} ✓
2. ${header[accept]} ✗ blank space as output
3. ${header["accept"]} ✓
4. ${header['accept']} ✓

Case-2: working with Bean:
${customer['name']}
${customer["name"]}
${customer[name]} blank space as o/p

case-3: Working with arrays:

```
<%
   String[] s ={"AAA","BB","CCCC"};
   pageContext.setAttribute("s",s);
%>
 <h1>
      ${s[0]}
      ${s["1"]}
      ${s['2']}
      ${s[3]}==>blank space
```

Case-4: working with List objects:

```
<%@ page import="java.util.*" %>
      <%
            ArrayList l = new ArrayList();
            l.add("AAA");
            l.add("BBB");
            l.add("CCC");
            l.add("DDD");
            pageContext.setAttribute("list",l);
            pageContext.setAttribute("index",3);
       %>
```

```
<h1>
    ${list[0]}<br/>
    ${list["1"]}<br/>
    ${list['2']}<br/>
    ${list[index]}
</h1>
```

Note: where ever property access operator is required there we can use collection access operator. But where ever collection access operator is required we may not use property access operator

3.Arithmetic operators:

1.+ operator:
EL does not support overloading and hence there is no string concatenation operator. "+" always acts as addition operator
Eg:
${2+3} ➔ 5
${"abc"+3} ➔ RE: NumberFormatException
${abc+3} ➔ 3
${""+3} ➔ 3
${null+3} ➔ 3
Note:
1. In Arithmetic operators null is treated as "0"
2. Empty String is also treated as "0"

2. - operator:
All rules are exactly same as + operator
${10-3} ➔ 7
${"10"-3} ➔ 7
${"abc"-3} ➔ RE:NFE
${abc-3} ➔ -3

3. * operator:
All rules are exactly same as + operator

${10 *3} ➔ 30
4. / operator or (div)
All rules are exactly same as + operator except that division operator always follows floating pointarithmetic
${10 / 2} ➔ 5.0
${10 div 2} ➔ 5.0
${10 / 0} ➔ Infinity
${0 / 0} ➔ NaN

5. % operator (or) mod operator:
In the case of mod operator both floating point and integral arithmetic are possible.
${10%0} ➔ ArithmeticException
${10%3} ➔ 1
Relational operators:
> or gt
< or lt
== or eq
>= or ge
<= or le
!= or ne
${10>3} ➔ true
${abcd==abc} ➔ true

Logical Operators:
There are 3 logical operators
& or && or and
| or || or or
! or not
Eg:
${!true} ➔ false
${true && false} ➔ false

Conditional operator:
${(3<4)?"yes":"no"} ➔ yes

Empty Operator:
${empty object}
true ➔ if object does not exist or
if object is an empty array
if object is an empty collection

if object is an empty string
otherwise returns false.

Eg:

${empty abc} ➔ true

${empty "abc"} ➔ false

${empty null} ➔ true

EL operator precedence:

1. unary operators (!, empty)
2. *,/,%,+,-
3.Relational
4.logical operators
5.conditional operator

EL Reserved Words:

1.true
2.false
3.null
4.empty
5.instanceof
6.gt
7.lt
8.ge
9.le
10.ne
11.eq
12.and
13.or
14.mod
15.div
16.not

EL versus null:

EL behaves very nicely with null.

1. In arithmetic operations null is treated as zero
2. In String evolutions null is treated as empty String
3. In Logical operators null is treated as false.

Eg:

${10+null} ➔ 10

${empty null} ➔ true

${!null} ➔ true