JSP

Date :- 14th may 2012 Mon

① Introduction
   ① servlets vs JSP.
   ② JSP lifecycle.

② JSP elements
   ① Directives [page, include, taglib]
   ② scripting elements.
   ③ Actions.

③ JSP scripting elements:
   ① Declarations
   ② scriptlets
   ③ expressions
   ④ comments.

④ JSP implicit objects

request
response
out
session
exception
config
application
page
pageContext.

⑤ JSP scopes
   page
   request
   session.
   application.

⑥ JSP Actions
   <jsp:useBean>
   <jsp:setProperty>
   <jsp:getProperty>
   <jsp:param>

<jsp:include-->
<jsp:forward-->
<jsp:plugIn-->
<jsp:fallback-->
<jsp:params-->
<jsp:declarations-->
<jsp:scriptlets-->
<jsp:expression-->

⑥ Custom Actions
① classic tag lib
② simple tag lib
— jsp —

⑦ JSTL
① core tags
② xml tags
③ fmt tags
④ SQL tags
⑤ Function tags

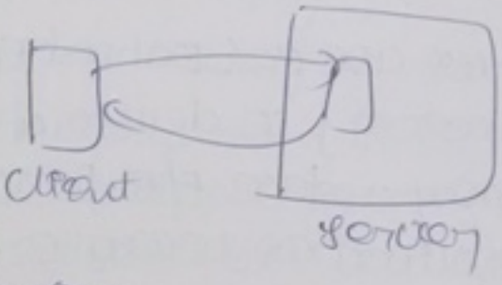
⑧ EL
① Introduction
② operators
③ Implicit objects
④ EL functions

Introduction

- In general in web applications, there are two types of responses.
  - (i) static response
  - (ii) dynamic response.

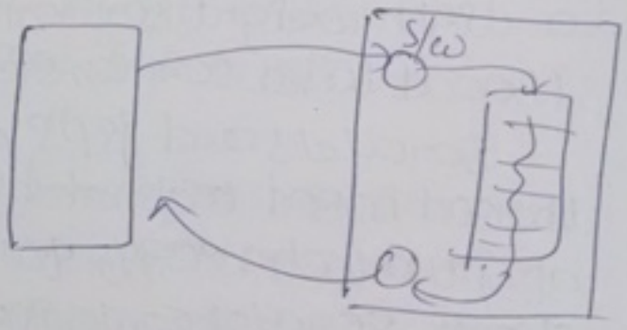
Static response: Static response is a response generated by the server without performing any action or without executing any server side resource.

Dynamic response:-



is a response generated by the server by performing an action or by executing a particular server side resource at server machine.

- To generate dynamic response from server machine, we have to execute a particular server side resource, for this we have to provide a server side application that is web application.



- To prepare a web application, we need a set of server side technologies called as web technologies like CGI, Servlets, JSP, PHP, Perl,

---

- Therefore the main requirement of cgi, Servlets, JSP, php, perl — is to design web applications in order to generate dynamic response from server.
- To design web applications if we use cgi technology then for every request server will prepare a separate process because CGI technology was designed on the basis of C technology that is a process based technology.
- If we use cgi technology as the server side technology to design web application then it may reduce the performance of server side application because cgi container may create more number of process when we increase number of requests.
- To overcome the above problem, we have to use a light weight server side technology, that is thread based server side technologies.

Servlets and JSP's are server side technologies, thread based technologies designed on the basis of java technology, for every request from clients, servlet container and jsp container will prepare a separate thread instead of process, to handle requests.

Here if we increase no. of requests even, the respective container will generate no. of threads only, not processes, This approach will increase the performance of server side

application.

2→

The main intention of introducing JSP technology is to reduce java code as much as possible in web applications.

If we want to design any web application by using servlets then we must require very good java knowledge. But if we want to design the same web application by using JSP's then it is not at all required to have java knowledge.

—In web application development it is not possible to design the web applications without java knowledge if we use servlets but by using jsp technology we are able to design the web applications without java knowledge. The main utilization of servlets in web applicat is to pick up the request and process the any request and to implement some the business logic at server side.

In web application development the main utilization of jsp technology is to prepare dynamic response to the client with very good look and feel that is to prepare presentation part.

— Servlet is a server side technology, it was designed purely on the basis of Java api, but Jsp technology is a server side technology, it was designed on the basis of servlet api and Java api.

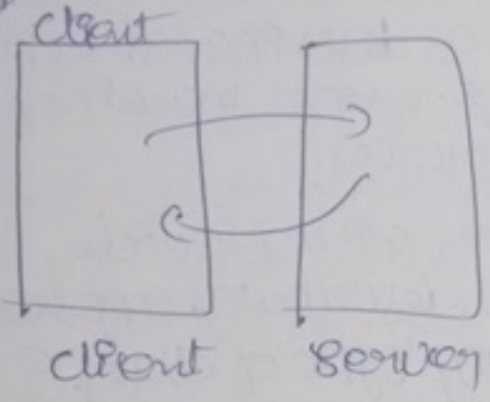
Note: In web applications, if we send any request to a particular Jsp page from client then Jsp page will be translated into servlet, by the execution of the translated servlets only, the required response will be generated to the client.

Therefore, Jsp pages will execute on the basis of servlets.

Imp. question

—what are the differences b/w web applications & distributed applications

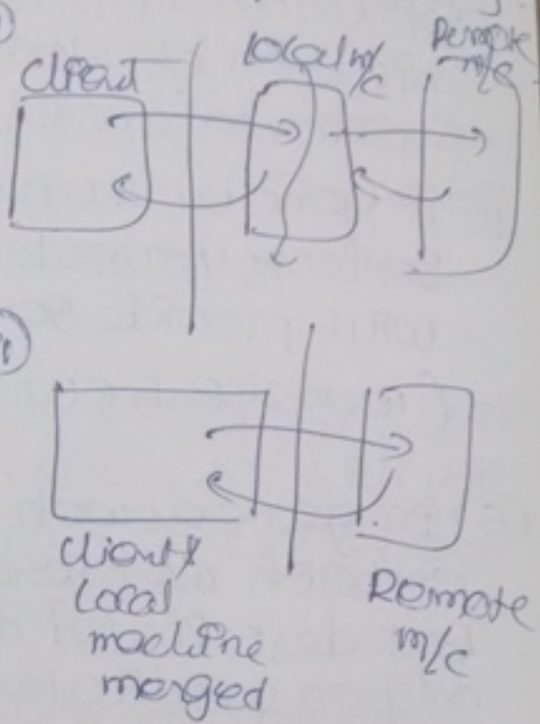
① web



web application

web application is a server side application which will be designed without distributing its application logic over multiple no. of jvm's.

② ⑪ ⑪ distributed applications



Distributed application is a server side application, it will be designed by distributing its application logic over multiple no of jvm's.

② —In general web applications will be designed by using a set of server side technologies called as web technologies like cgi, servlets, jsp's and so on--

—Distributed applications will be designed by using a set of technologies called as distributed technologies like socket programming, rmi, ~~corba~~ EJB's, webservices and so on-

③ The main purpose of web applications is to generate dynamic response from servers machine.

But the main purpose of distributed applications is to establish a distributed communication b/w local machine & remote machine in order to access remote services.

④ In general web applications will provide services for web clients but distributed applications will provide services for any type of clients any (web clients, CUI application program, EDI application program)

⑤ In general web applications will be executed on both web servers and application servers. But distributed applications will be executed only on application servers.

⑥ Web application is a collection of web components it will be executed by using web containers like servlet container to execute servlets, jsp container to execute jsp's and so on--

⑦ Distributed application is a collection of distributed components, it will be executed by using distributed containers like EJB Container to execute EJB's ✓

Diff b/w Servlets & JSP's

Date: 15th May, 2012 Tue

— If we want to design web application on the basis of MVC architecture or MVC design pattern where we have to use a servlet as a controller and a set of jsp pages as a view part.

— Struts is a MVC based framework, where we will use ActionServlet, a type of servlet as controller and a set of jsp pages will be used as view part.

— JSF (Java Server Faces) is MVC based framework to design web applications, where we will use FacesServlet, a type of servlet as controller, and a set of jsp pages as view part.

— In case of the servlets, we are unable to separate presentation logic and business logic but in case of jsp pages, we are able to separate presentation logic and business logic due to the availability of separate set of tags for presentation and separate set of tags for business logic. or,

Note: In case of jsp pages, it is possible to use <html> tags to prepare presentation logic and we will use jsp tags to prepare business logic.

2 →

1→ If we perform any modifications on existed Servlet then it is required to perform recompilation and reloading on to the server.

If we perform any modifications on any JSP pages then it is not required to recompile and it is not required to reload on to the server because the JSP pages are auto compiled and auto loaded.

2→ In general, we will keep the JSP pages under application folder, but it is possible to keep the JSP pages at any location of the web application directory structure.

If we keep the JSP pages under application folder, that is public area then client is able to access that JSP page by using its name directly in the url.

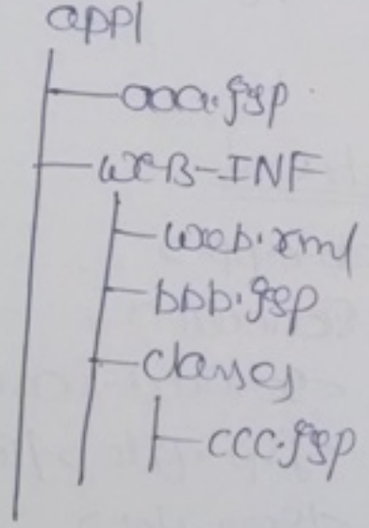
If we deploy the JSP page under private area that is WEB-INF folder and its internal locations then we have to configure, that JSP page in web.xml file with a particular url pattern; with this url pattern only we are able to access that JSP page from client. To configure a JSP page in web.xml file, we have to use the following xml tags.

```
<web-app>
  ==
  ===
<servlet>
  <servlet-name> logical name </servlet-name>
  <jsp-file> context relative path of the jsp page </jsp-file>
</servlet>
<servlet-mapping>
  <servlet-name> logical name </servlet-name>
  <url-pattern> pattern-name </url-pattern>
</servlet-mapping>
  ===
</web-app>.
```

Exple

aaa.jsp

```
<html>
  <body>
   <center><b><font size="7"
              color="red">
   <br><br>
First Jsp application from
      application folder.
</font></b></center></body>
</html>
```



bbb.jsp

```
<html>
  <body>
    <center> <b> <font size="7" color="red">
      <br><br>
First Jsp application from WEB-INF folder.
  </font> </b> </center> </body> </html>.
```

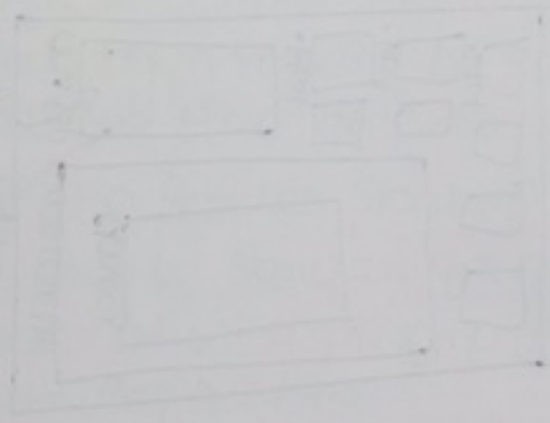
ccc.jsp

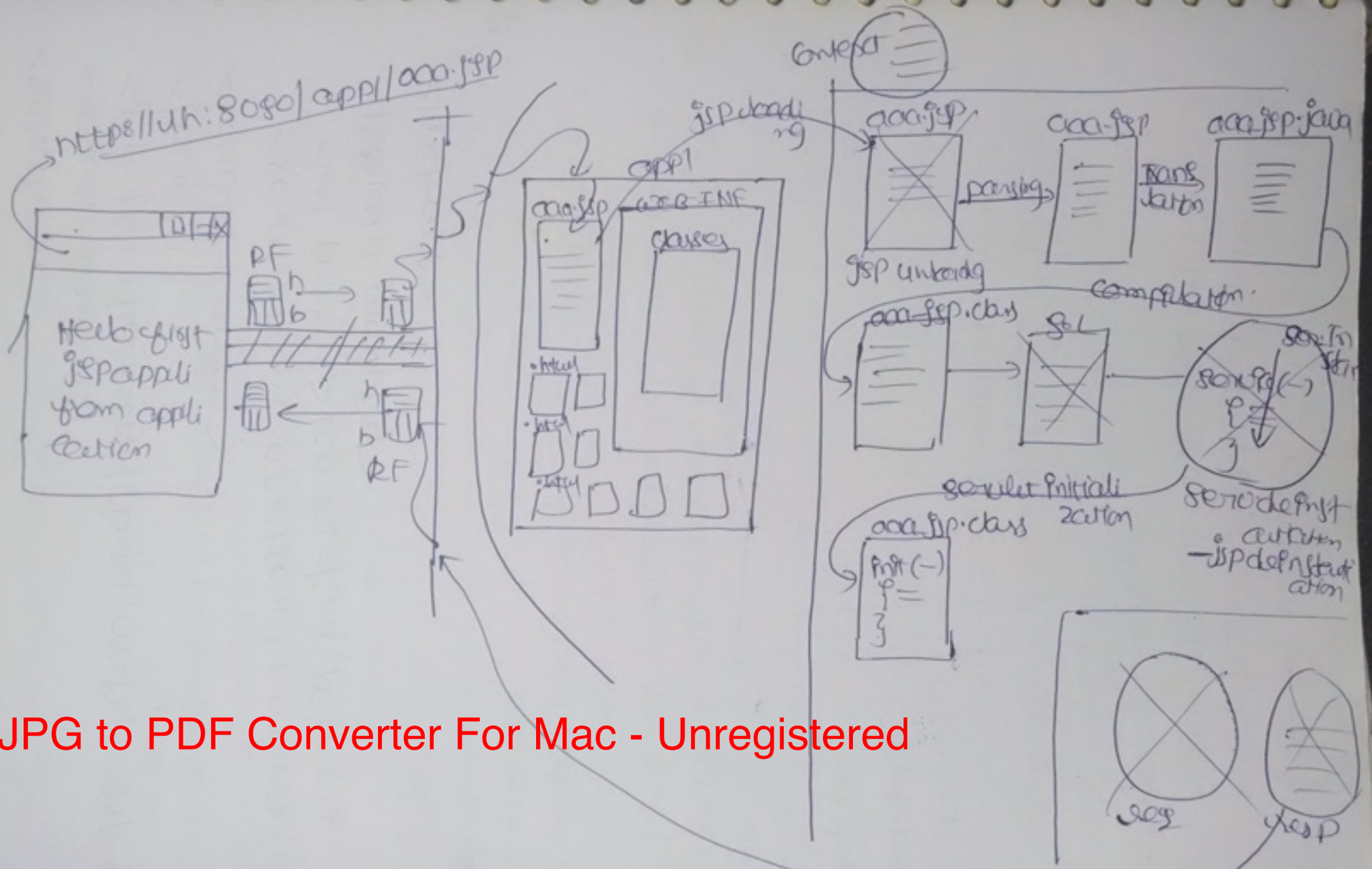
```
<html>
 <body>
 <center><b> <font size='7' color='red'>
      <br><br>
 First Jsp application from classes folder.
  </font></b> </center> </body> </html>.
```

web.xml

```
<web-app>
 <servlet>
   <servlet-name> bbb </servlet-name>
   <jsp-file> /WEB-INF/ bbb.jsp </jsp-file>
  </servlet>
   <servlet-mapping>
     <servlet-name> bbb </servlet-name>
     <url-pattern> /wobint </url-pattern>
     </servlet-mapping>
```

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





Date: 16th may, 2012 wed

From the above representation,
When we start server, automatically container will recognize each and every web application which are available in web-apps folder and prepare separate Servlet Context object for each and every web application.

2→ At the time of recognizing web application container will identify web.xml file, perform web.xml file loading, parsing and reading the content, if container identify any application level data then it will be stored in Servlet Context Object at the time of creation.

2→ After the server startup if you send a request from client to server for a particular JSP page then protocol will pickup the request and perform the following actions.

① establish a virtual socket connection from client to server as per the server ip address and port number which we provided.

② protocol will prepare a request format with header part and body part, where header part will manage all the request headers and body part will manage all the request parameters provided by user.

③ After getting request format protocol will carry that request format to server.

2 → when request is reached to server main server will pick up the request and check the request data is in well formed format or not.

If the request data is in well formed format then that request will be sent to container. Container will pick up the request, identify the application name and resource name from url. If the resource name is a jsp page then container will search for it under application folder.

2 → when container identify the requested jsp page under application folder then container will perform the following actions.

① Jsp loading :- Here container will load the requested jsp page into the memory from the web application folder structure.

② Jsp parsing :- Here container will check all the syntaxes which we have used in jsp page are valid or not.

③ Jsp translation to Servlet :- After the jsp parsing container will convert that jsp page to a servlet.

What is parsing :-

When a jsp page is translated to a servlet then that servlet respective .java file will be available in the server software (Tomcat server) at the following location.

C:\Tomcat 7.0\work\Catalina\localhost\app1\

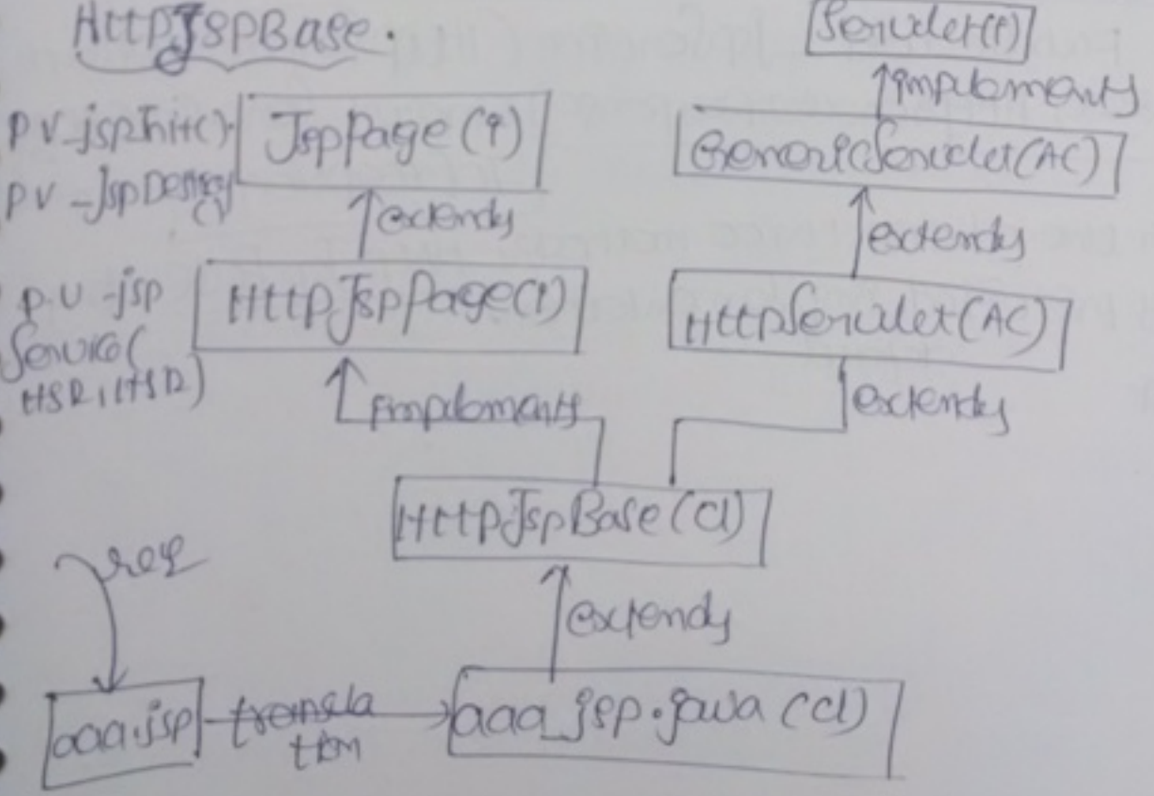
folder name

user application name

org\apache\jsp\aaa_jsp.java

equivalent servlet code to the user defined aaa.jsp page.

2→ Tomcat server has provided aaa_jsp.java class as final class. The default super class for the translated servlet is HttpJspBase.



— when we send a request to aaa.jsp, then container will prepare aaa_jsp.java file, where container will override _jspService(-) method of HttpJspBase class in aaa_jsp.java by including the content what we provided inside aaa.jsp

2→

```
public void _jspInit()
public void _jspDestroy()
```

→ where HttpJspPage interface has declared the following method

```
public void _jspService(HttpServletRequest req, HttpServletResponse) throws ServletException or IOException
```

For the above three methods, HttpJspBase class has provided ~~x~~ default implementation.

④ Translated Servlet Compilation :- After getting Servlet.java file, Container will compile it and generate the respective .class file.

⑤ Servlet loading :- After getting the translated Servlet.class file, Container will load its byte code to the memory.

⑥ Servlet instantiation :- Here Container will create an object for the loaded Servlet.

⑦ Servlet initialization :- Here Container will execute _jspInit() method to prepare Servlet initialization.

⑧ creating request and response objects :- After completing the Servlet initialization Container will create a thread to access _jspService() method. For this Container will prepare HttpServletRequest object and HttpServletResponse object.

⑨ generating Dynamic response :- By passing request and response objects to _jspService() method, Container will execute _jspService() method and generates the dynamic response in response object.

⑩ Dispatch dynamic response to client :-
when the Container generated thread reached to the ending point of _jspService()

method then that thread will be in dead state, with this container will bypass the generated dynamic response to the main server, where main server will bypass that response to protocol. where protocol will send that response to client by preparing response format.

(11) Destroy request and response objects:-

when the dynamic response reached to client browser, then protocol will terminate the virtual socket connection which we established, with this container will destroy the request and response objects.

(12) Servlet Deinstantiation:- After destroying request and response objects, container will be in waiting state, depending on the container implementation, if container identifies no further request to the same resource then container will destroy servlet object. For this container will execute -JSP Destroy() method.

(13) Servlet unloading and JSP unloading:-

After the servlet Deinstanciation, container will eliminate the servlet byte code and JSP byte code from the operational memory.

The above jsp life cycle will be divided into the following two phases.

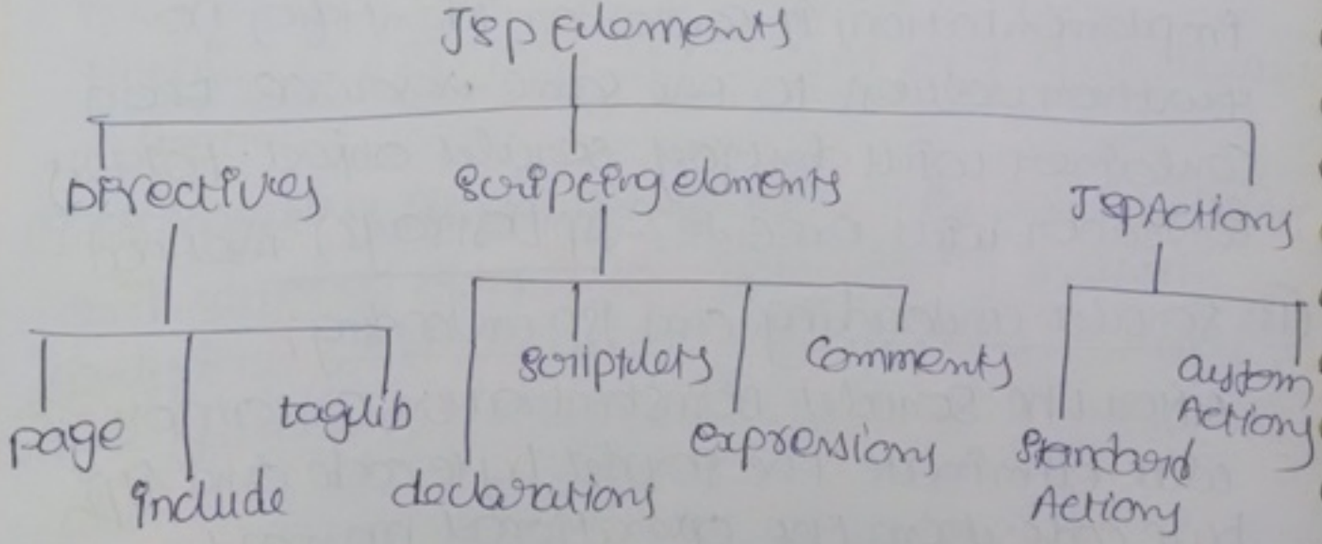
① Translation phase.
② Request processing phase.

① Translation phase:- This phase will include jsp loading, jsp parsing, jsp translation to servlet and the translated servlet compilation.

② Request processing phase:- This phase includes the jsp life cycle stages right from servlet loading to jsp unloading.

Date:- 17th may, 2012 Thurs

- To design jsp pages in web applications, we have to use the following elements in jsp pages.



What are the diff b/w JSP directives and scripting elements?

① JSP directives can be used to define present JSP page characteristics, to include the target resource content into the present JSP page content, and to make available user defined tag library into the present JSP page.

JSP scripting elements can be used to provide Java code inside the JSP pages.

② In general, almost all the JSP directives will be evaluated at the time of translating JSP page into a servlet.

JSP scripting elements will be evaluated at the time of request processing in JSP execution.

③ In JSP technology majority of the JSP directives will not give direct effect to response generation but almost all scripting elements except comments will effect the response generation.

Int: To design JSP pages, we have already scripting elements, what is the requirement to go for JSP Actions?

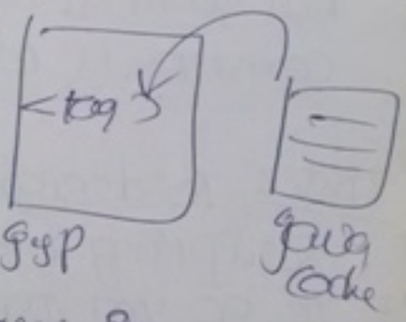
— In JSP technology, by using scripting elements we are able to provide Java code inside the JSP pages, but the main theme of the JSP technology is to reduce Java code inside the web applications that is inside the JSP pages.

In the above context to preserve the theme of the JSP technology, we have to eliminate java code from JSP pages, to eliminate java code from JSP pages, we have to eliminate scripting elements from JSP pages.

—To eliminate scripting elements from JSP pages we have to use an alternative provided by JSP technology that is JSP Actions.

In case of JSP actions, we will define a scripting tag in place of java code in JSP page, with respect to the scripting tag the required java code must be provided internally.

In the above context, when JSP container encounter the respective scripting tag automatically container will execute internal java code. By this an action will be performed by the container called as JSP Action.



2 real time Assume I am a team lead, in a company. Suddenly there is a requirement to recruit 10 members in an java project. My intention is to I don't recruit any other b.tech or mca or m.tech students because I have to give more salaries to those guys. So I decided to recruit B.Sc students who don't know any thing about java because I will give only 5000 to each recruited graduate employee. My project development

so recruit of BSc people know anything

cost will be reduced. In part of the first 10 days of training period, I will give traing on html technology. and immediately I will start my project (java project).

Suppose in my java project, we will have a requirement to design totally 150 number of web pages. In these 50 number of web pages including javacode have to interact with the database.

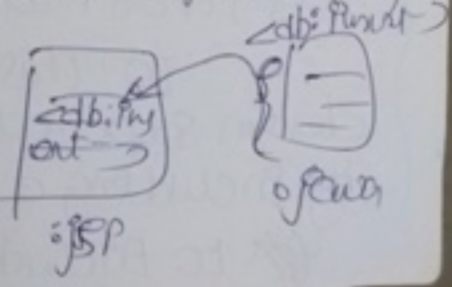
So first of all, I will assign the remaining 100 web pages designing work to the 10 employees who were recruited.

As a java programmer, I will design the 50 web resources having java code in that pages design. and I will attain separate tags for each and every .java file. for ex:

<db tag: insert → tag for a java file which do inserting of data into database.
.java

<db tag: update → tag for a java file which do update records of a table in database.
.java

2→ For the 10 employees, if they have any requirement to use any of the 50 java pages, they simply write the tag associated with that java page in their page designing. so that at the time of execution, simply that java code replace in the tag.



<db tag: insert update one java files prop are java code 150 web pages 50 requi re db interacting they will prepare they

Jsp Directives

There are three directives are defined in jsp technology

1. page directive
2. include directive
3. taglib directive

page directive: — In jsp technology, page directive can be used to define present jsp page characteristics like importing a package, specifying a particular super class to the translated servlet, to make available jsp implicit objects like session, exception and so on.

Syntaxes: There are two types of syntaxes for jsp directives

1. JSP syntax.
2. XML syntax.

JSP syntax: `<%@ page [Attribute-list] %>`

xml syntax: `<jsp:directive page [Attribute-list] %>`

So, when will there jsp scripting elements useful?

Ans Suppose I have a requirement to write less number of java code in my designing page, then I will use jsp scripting elements.

For this requirement, it is not good to design separate java resource files and use them by including a tag. all that is unnecessary to include two or 3 lines of java code.

Using of JSP scripting elements is against to the JSP rules, still why are they using? Why don't they remove from JSP?

Ans Except scripting elements, if u use all other the elements of JSP, in order to design a web project, you may not meet the 100% requirements so inorder to meet the complete requirements you have to use scripting elements.

— JSP page directive may include the following list of attributes.

Date: 18th may, 2012 Fri

1. language
2. content type
3. import
4. extends
5. info
6. errorpage
7. isErrorpage
8. Buffer
9. autoflush
10. session
11. isThreadSafe
12. isELIgnored

language:— The main purpose of this attribute is to specify a particular language in order to use in the scripting elements.

- If we want to specify any particular language as value to language attribute then first name of all we have to make sure whether the underlying server support the specified language
- The default value of this attribute is of not java.

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

The other purpose of scripting elements is to allow other types of languages also into the page (like cobol, —).

- contentType:- This attribute can be used to specify a particular MIME type like text/html, text/xml, image/jpeg, application/pdf, — inorder to give an information to the client about to specify the type of response way the server the resource generated.
- The default value of this attribute is text/html.

Ex: <%@page contentType="img/jpg"%>

Import :- This attribute can be used to import a particular package into the present jsp page. The default value of this attribute is

java.lang, javax.servlet, javax.servlet.http, java.io, javax.servlet.jsp

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

2 → If we want to import multiple number of packages in present jsp page either we have to provide multiple no of packages with comma separator as a value to single import attribute and by repeating import attribute for each and every package.

① <%@page import="java.util.*, java.net.*"%>

② <%@page import="java.util.*"
import="java.net.*"
import="java.sql.*"%>

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

③

Extendy :-

This attribute can be used to specify a particular class as a super class to the translated servlet.

The default value of this attribute is HttpJspBase

- If we want to specify any particular class name as super class to the translated servlet then that class must be an implementation class to HttpJspPage interface and must be a sub class to HttpServlet or a direct sub class to HttpJspBase class

⑤ Info :-

This attribute can be used to specify the generalized description about the present jsp page. (It is equal to the comment in .java file)

The default value of this attribute is

(we are using Jasper Jsp 2.2 Engine apache software foundations provide Tomcat server which contains jsp 2.2 version)

Ex:-

```
<%@page info="First JSP application" %>
<html>
<body>
<center><b><font size="7" color="red">
<br><br>
```

<%= getServletInfo() %> → It's a scriptlet tag element

 </body></html>

↳ Save this page as info.jsp in webapps/app1/ folder

execution:

—If we specify any value to info attribute in page directive then we have to use getServletInfo() method in jsp page to get the info attribute value.

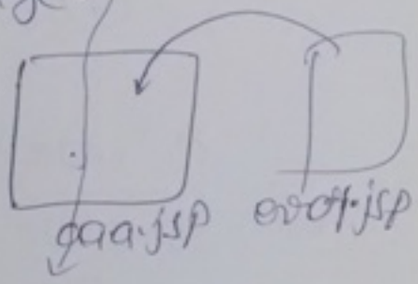
—In the above

1/①
Jsp
22
Aug

—errorpage:-

The main purpose of errorpage attribute is to specify the name and location of an errorpage in order to execute when we get an exception in the present jsp page.

—no default values are existed for this attribute.



isErrorPage :-

It is a boolean attribute. It can be used to give an information to the container about to allow or not allow exception implicit object in the present jsp page.

- The default value of this attribute is false.

exp aaa.jsp

```
<%@page errorpage="error.jsp" %>
<html>
<body>
<center><b><font size="7" color="red">
<br><br>
<%
java.util.Date d=null;
out.println(d.toString());
%>
</font></b></center></body></html>
```

↳

error.jsp

```
<%@page isErrorPage="true" %>
<html>
<body bgcolor="lightyellow">
<center><b><font size="7" color="red">
<br><br>
<%=exception %>
</font></b></center></body></html>
```

session:— It is a boolean attribute, it can be used to give an information to the container about to allow or not to allow session implicit object. In the present jsp page.
— the default value of this attribute is true
(That means by default the session object available to the jsp page when we use this attribute in our jsp page)

isThreadSafe:—

It is a boolean attribute, it can be used to give an information to the container about to allow or not to allow multiple number of requests at a time to the present jsp page.
— The default value of this attribute is true.
what is threadsafe resource?

Q18. Syntax:

<%@page isThreadSafe="true"%>

— If we provide isThreadSafe value is true then the translated servlet will not implement single thread model interface inorder to allow multiple no. of requests at a time.

— If we provide isThreadSafe attribute value false then the translated servlet will implement singleThread model interface inorder to allow only one request at a time.

notes: In web applications all the servlets and JSPs are by default thread safe.

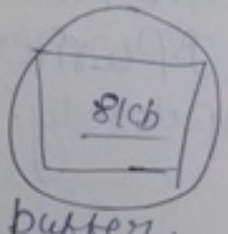
note: In java applications, if any resource is able to process multiple number of requests that is threads at a time without having data inconsistency then that resource is called Thread safe resource.

buffer attribute:

Date: 20th may 2012 Sun

This attribute can be used to specify a particular size to the buffer available in JspWriter object.

The default value of this attribute is 8kb

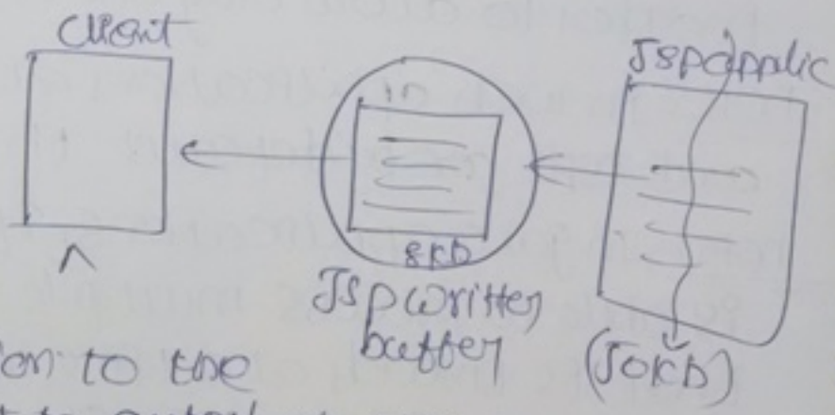


buffer.

Note: In Jsp technology, to send some response to client we will not use printWriter object because printWriter is not buffered writer, so that printWriter will reduce the performance Jsp application. Due to this reason, Jsp technology has used a separate writer that is: JspWriter, it is a buffered writer, it will improve the performance of Jsp applications.

ex: <%@page buffer="50KB" %>

Autoflush :-



autoflush is a boolean attribute; it can be used to give an intimation to the container, about to autoflush the response to client or to raise an exception, when Jsp writer buffer filled with the response.

If we provide autoflush attribute is true then container will dispatch the response from JspWriter buffer to client when it reach the maximum capacity of the buffer.

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

exception: org.apache.jasper.JasperException.
root cause: java.io.IOException: Error:
JSP Buffer Overflow.

example
p4s

```
<%@page buffer="52kb" autoflush="false" %>
<html>
  <body>
  <center><b><font size="7" color="red">
  <br><br>
  <%
  for(int i=0; i<100000; i++)
    {
      out.println("AFFA");
    }
  %>
  </font></b></center></body></html>
```

↳ save this one as appl.jsp in D:\Tomcat 7.0\webapps\appl

→ now open browser and type the url as following in the address bar.
https://localhost:9090/appl/appl.jsp
⟹ then an exception occured on the browser contains the content as mentioned above.

- The default value of autoFlush attribute is true

Note:

① If we provide buffer attribute value 0kb and autoFlush attribute value is false, then container will raise an exception like exception: org.apache.jasper.JasperException: /aa.jsp(12) jsp.error.page.badCombO

isELIgnored :-

It is a boolean attribute. It can be used to give an information to the container about to allow or not to allow expression language syntaxes in the present JSP pages are not.

If we provide true as value to isELIgnored attribute then container will not allow expression language syntaxes in present jsp pages.

If we provide false as a value to isEL attribute then container will Ignored allow expression language syntaxes into the present JSP pages.

- The default value of this attribute is false

Include directive

The main purpose of the include directive in jsp pages is to include the target resource content into the present jsp page content.

2→ There are two types of syntaxes for include directive

① Jsp syntax.
② xml based syntax.

① Jsp syntax:

```
<%@ include file="—" %>
```

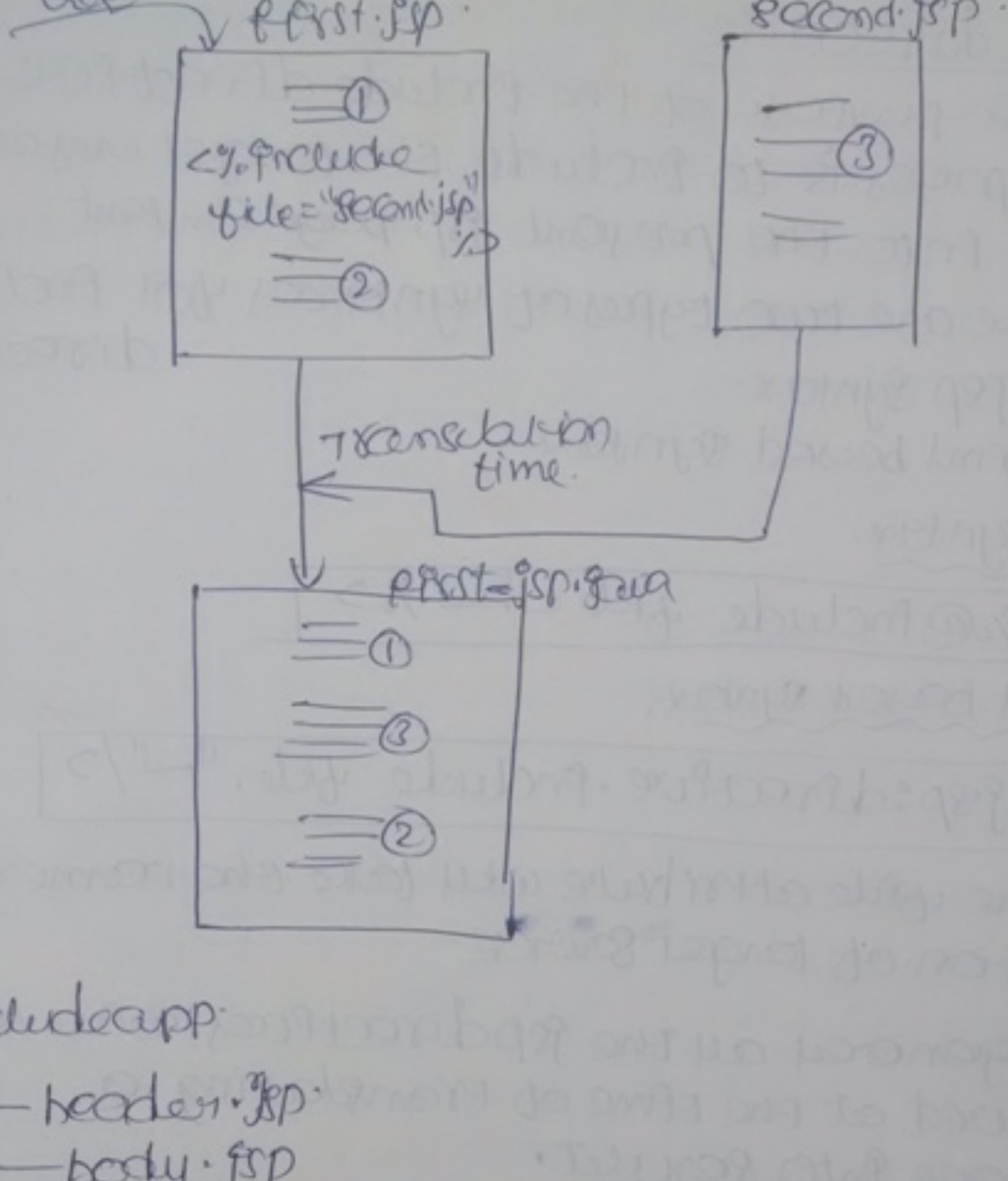
② xml based syntax:

```
<jsp:directive.include file:"—"/>
```

— where file attribute will take the name and location of target source.

2→ In general all the jsp directives will be resolved at the time of translating a jsp page into servlet.

If we provide include directive in any jsp page the target resource content will be included than in the present jsp page at time of translation.



includeapp

- header.jsp
- body.jsp
- footer.jsp
- home.jsp
- WEB-INF
  - classes

header.jsp

```
<html>
  <body>
  <center><b><font size="7" color="red">
    Durga software solutions.
    </font></b></center></body></html>
```

body.jsp

```
<html>
 <body>
 <center><b><font size="6">
  <p>
   <br><br>
 Durga software solutions is very good real
 training center. Dss is well proven free
 of certification training.
 </p></font></center></body></html>
          <b>
```

footer.jsp

```
<html>
  <body>
   <center><b><font size="6" color="red">
  copy rights 2010-2011@cocco.durgasoft.com.
  </font></b></center></body></html>
```

home.jsp

```
  <html>
   <body bgcolor="lightgreen">
   <% include file="header.jsp" %>
    <br><hr><br>
```

```
<%@ include file="body.jsp" %>
    <br><br><hr><br>
<%@ include file="footer.jsp" %>
</body></html>
```

==

taglib directive:- The main purpose of taglib directive is to make available user defined tag library into the present JSP page.

Syntax:

<%@ taglib uri="___" prefix="___" %>

where uri attribute will take an uri value to represent the location where user defined tag library is available.

where prefix attribute will take prefix name to the custom tags which we are going to use in the present JSP page.

Note:- In general, we will utilize taglib directive to the custom tags design to make available tag implementation class to JSP page.

<jsp:useBean

Scripting elements

The main purpose of scripting elements in JSP technology is to allow java code inside the JSP pages.

There are 3 types of scripting elements.

1. Declarations
2. Scriptlets
3. Expressions

declarations: This scripting elements in the JSP page can be used to allow all the java declarations inside the JSP page.

Syntax: <%!
= java declaration.
%>

If we provide java declarations by using declarations scripting element then the provided java declarations will be available in the translated servlet in outside of _jspService() method.

Scriptlets:

This scripting element can be used to provide a block of java code inside JSP page.

<%
= block of java code
%>

If we provide a block of java code with scriptlets in JSP page then that block of java code will be available in the translated servlet under _jspService() method.

③ Expression :-

This scripting element can be used to evaluate a single java expression and provide the result in the response object inorder to display on to the client browser.

Syntax: <%= expression %>

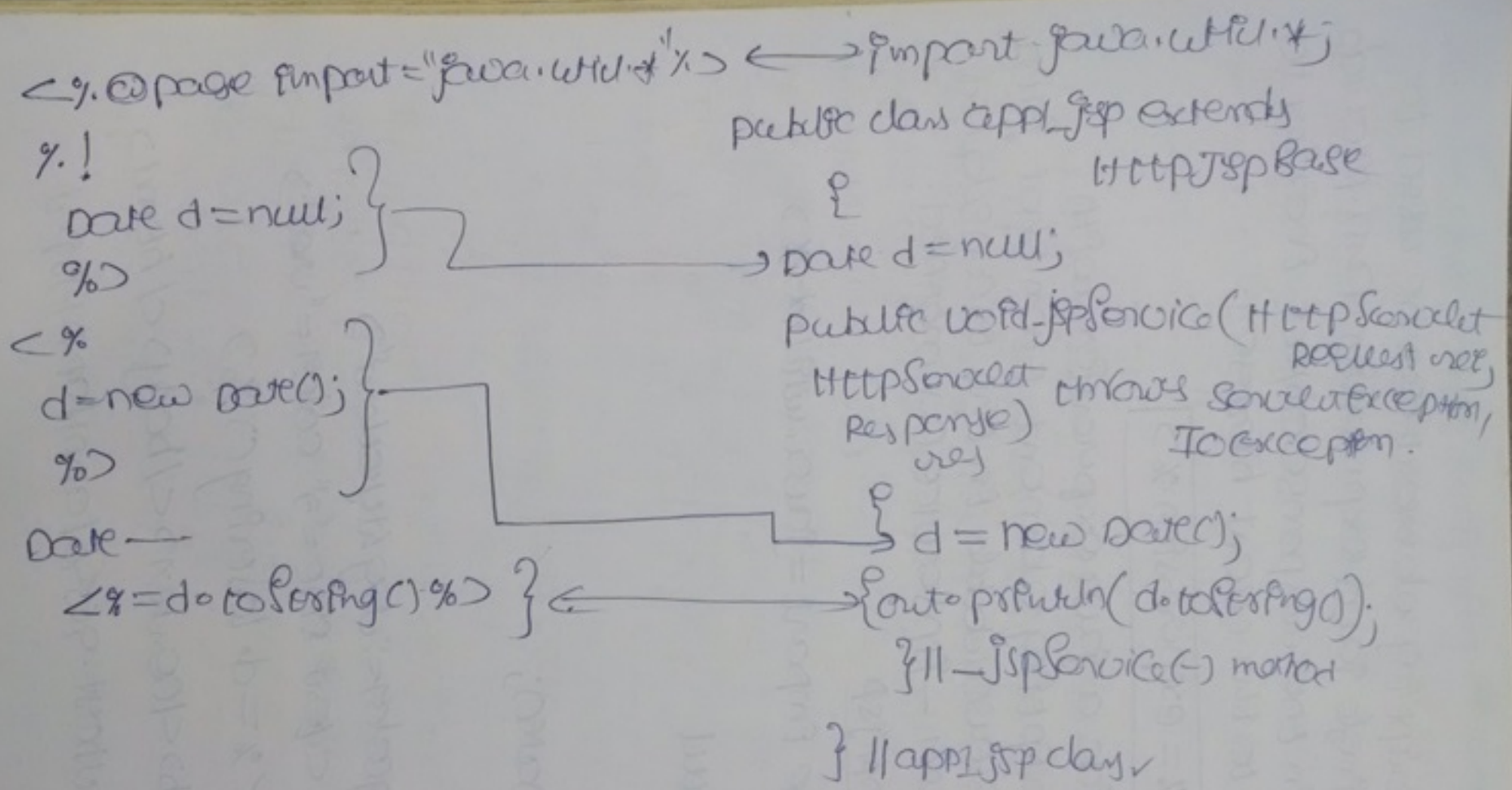
If we provide a java expression with expression scripting element then that java code will be available in the translated servlet under _jspService(-) method.

example app1/app1.jsp

Program

```
<%@ page import=java.util.* %>
<%!
Date d=null
%>
<%
d=new Date();
%>
<html>
<body bgcolor="lightyellow">
<center><b><font size="5" color="red">
Date --- <%= d.toString() %>
</font></b></center></body></html>
```

— http://localhost:9090/app1/app1.jsp



JSP Implicit objects

— In JSP applications, it is frequent requirement for the developer to display some data on command prompt. To perform this, if Java technology has not provided any predefined support then Java developers has to prepare printStream object explicitely.

— once printStream object is frequent requirem for each and every Java developer then out all the Java developers may expect the required printStream object as predefined object. In the above context, to fulfill all the developers requirement Java api has provided the required printStream object as predefined object in the form of out variable in System class.

2 → similarly in the web applications we may use some objects frequently like printWriter, Request, Response, config, Context, Session and so on in web applications.

To get all the above specified frequent used objects in our web applications we have to use some piece of Java code.

Due to the above reason, JSP technology has provided all the frequent used objects as predefined, JSP predefined objects are called as JSP implicit objects.

— In JSP technology with the introduction of implicit objects it is possible to eliminate a little bit of Java code from JSP pages.

— JSP technology has provided the following list of implicit objects.

Implicit objects	Types
out	javax.servlet.jsp.JspWriter
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
config	javax.servlet.ServletConfig
application	javax.servlet.ServletContext
page	java.lang.Object
exception	java.lang.Throwable
session	javax.servlet.http.HttpSession
pageContext	javax.servlet.jsp.pageContext

Int. Qu: what is the difference b/w printWriter and JspWriter

printWriter is a writer object in java.io package. It can be used to carry the response from a servlet to response object.

printWriter is not buffered writer so that printWriter will provide less performance in web applications.

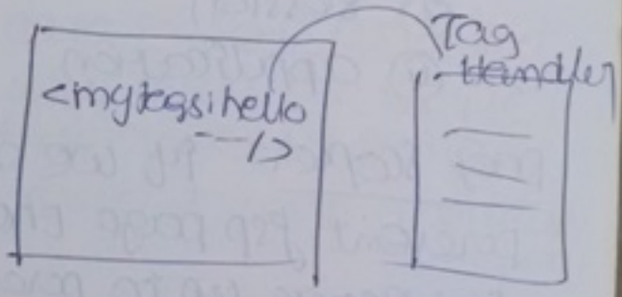
JspWriter is a writer object provided by JSP technology as javax.servlet.jsp.JspWriter. It can be used to carry response from JSP page to the response object.

JspWriter is a buffered writer, it will provide a very good performance in web applications.

2 → what is pageContext? and what is the purpose of pageContext.

Ans pageContext is one of the JSP implicit object in JSP applications. pageContext can be used to make available all the JSP implicit objects in non-JSP environment.

Ex: In custom tags design, we have to define a separate java class as tag handler class for each and every JSP custom tag, where in the respective tag handler classes if we want to use JSP implicit objects then we have to use pageContext implicit object.



To return all the JSP implicit objects pageContext abstract class has provided the following methods.

```
public Xxx getXxx()
```

where xxx may be out, request, response, ---

Ans JspWriter out = pageContext.getOut();
HttpServletRequest req =
pageContext.getRequest();

Jsp Scopes :-

Make available the data up to some number of resources is called as scope.

— In J2SE applications, to define scope for the data Java technology has provided scope related access modifiers like public, protected, default and private.

— Similarly in web applications to define scope for the data JSP Technology has provided the following four types of scopes along with J2SE provided scope related access modifiers

① page
② request
③ session
④ application

page scope :- If we declare any data in the present jsp page then that data will have the scope up to present jsp page.

~~If we declare any data in~~

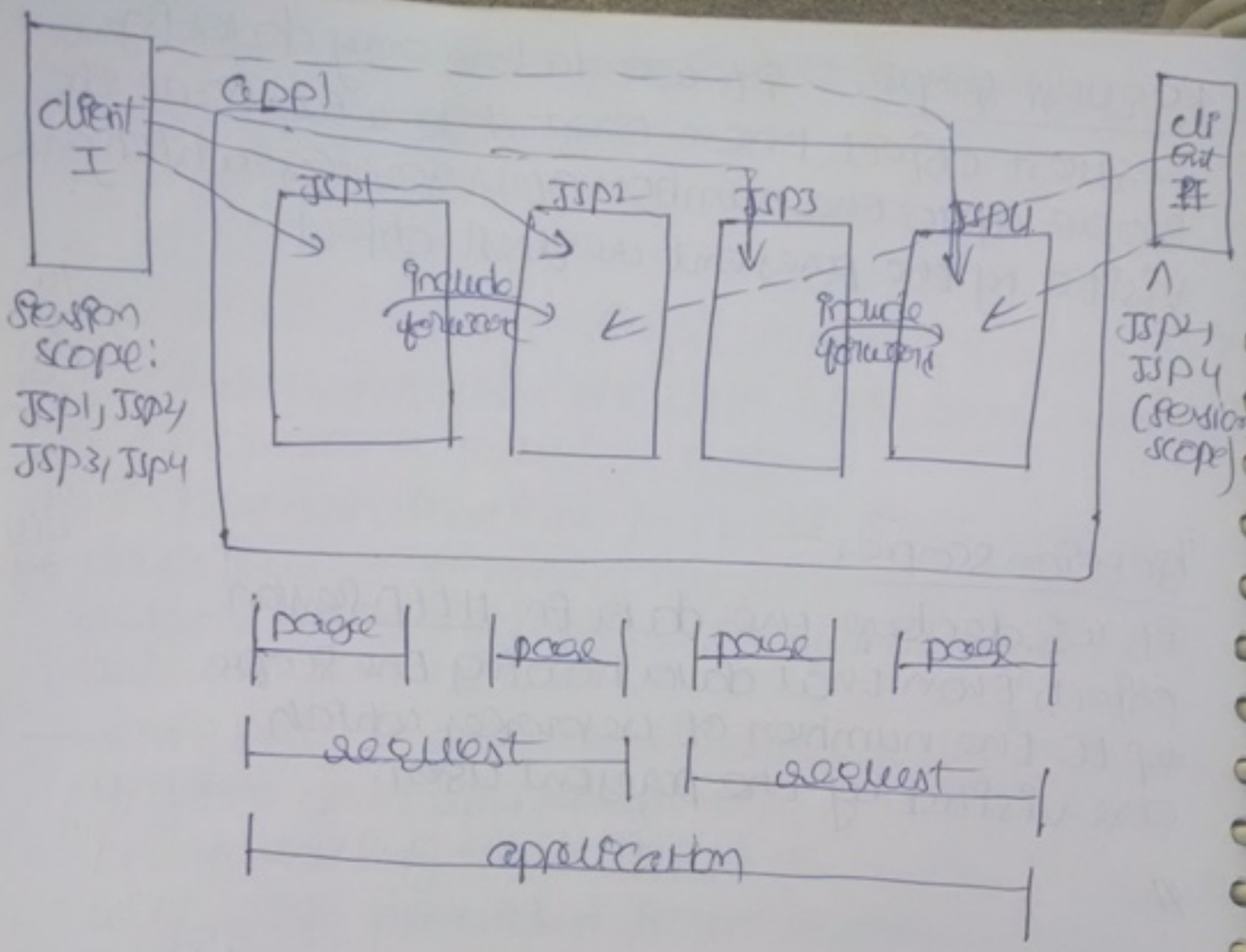
Request scope:- If we declare any data in request object then that data will have the scope up to the number of resources which are visited by the present request object.

Session scope:-

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

application scope:

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



appl

- empform.html
- empdisplay.jsp
- WEB-INF
  - classes

empform.html

```
<html>
  <head>
  <center><b><font size="7" color="red">
   Employee Registration form.
  </font></b></center></head>
   <br><br><br><br>
  <body bgcolor="lightpink">
   <b><font size="7">
  <form method="get" action="./empdisplay.jsp">
   <pre>
  Employee Id: <input type="text" name="eno"/>
  Employee name: <input type="text" name="ename"/>
Employee Salary: <input type="text" name="esal"/>
Employee Address: <input type="text" name="eaddr"/>
  <input type="submit" value="Register"/>
  </pre></form></font></b></body></html>
```

empdisplay.jsp

```
<html>
  <head>
  <center><b><font size="4" color="red">
      Registration Details.
  </font></b></center></head>
  <br><br><br>
  <body bgcolor="lightyellow">
   <b><font size="6">
Employee Number --- <%=request.getparameter
                      ("eno")%>
   <br><br>
Employee
 Name --- <%=request.getparameter("ename")%>
   <br><br>
Employee
 Salary ---> <%=request.getparameter("esal")%>
          <br><br>
Employee
 address --- <%=request.getparameter("eadd")%>
  </font></b></body></html>
```

app2

- addform.html
- addproduct.jsp
- success.jsp
- failure.jsp
- existed.jsp
- WEB-INF
  - lib
    - ojdbc6.jar
  - classes

addform.html

```
<html>
 <head>
   <center><b><font size="7" color="red">
    product Add Form.
   </font></b></center></head>
   <br><br><br><br>
 <body bgcolor="light blue">
   <b><font size="7">
 <form method="get" action="/.addproduct.jsp">
   <pre>
product-Id:<input type="text" name="pid"/>
product Name <input type="text" name="pname"/>
product Cost < input type="text" name="pcost"/>
 <input type="submit" value="ADD"/>
 </pre></form></font></b></body></html>
```

success.jsp

```
<html>
  <body bgcolor="lightyellow">
  <center><b><font size="7" color="red">
  <br><br>
  product Registration success.
  </font></b></center></body></html>
```

failure.jsp

```
<html>
 <body bgcolor="lightyellow">
 <center><b><font size="7" color="red">
   <br><br>
   product Registration failure.
   </font></b></center></body></html>
```

addproduct.jsp

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

```
Con=DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","system",
                                    "durga");
    st=con.createStatement();
pid=request.getparameter("pid");
pname=request.getparameter("pname");
price=Integer.parseInt(request.getparameter
                                    ("price"));
rs=st.executeQuery("select * from product
                    where pid='"+pid+"'");
  boolean b=rs.next();
    if(b==true)
    {
RequestDispatcher rd=application.getRequest
 Dispatcher("/existed.jsp");
    rd.forward(request,response);
  }
  else
  {
int rowCount=st.executeUpdate("insert into
 product values('"+pid+"','"+pname+"',"+price+
                    ")");
    if(rowCount==1)
    {
 RequestDispatcher rd=application.
  getRequestDispatcher("/success.jsp");
    rd.forward(request,response);
  }
```

```
else
{
RequestDispatcher rd = request.getRequest
        Dispatcher("/procedure.jsp");
    rd.forward(request response);
    }
}
}
catch(Exception e)
    { e.printStackTrace();
    }
%>
```

existed.jsp

```
<html>
<body bgcolor="lightyellow">
<center><b><font size="7" color="red">
<br><br>
product Already existed.
</font></b></center></body></html>
```

JSP Actions

Date: 21st May 2012 Mon

In JSP technology, by using scripting elements we are able to provide java code inside the jsp page. But the main theme of the jsp technology is not to allow java code inside the jsp pages. To preserve the theme of the jsp technology, we have to eliminate java code from jsp pages, for this we have to eliminate scripting elements from jsp pages.

2→ To eliminate scripting elements from jsp pages we have to use an alternative provided by Jsp technology that is JSPActions.

In case of JSPActions, we will provide a scripting tag in jsp page in place of java code and we will provide the same amount of java code under classes folder. when container encounter the scripting tag then container will execute the internal java code, By the execution of this java code container will perform an action called as JSPAction.

There are two types of JSP Actions

1. Standard Actions
2. Custom Actions

Standard Actions :- These are the predefined actions defined by jsp technology in the form of a set of tags. that is action tag.

- Jsp technology has provided all the action tags in the form of xml syntax.

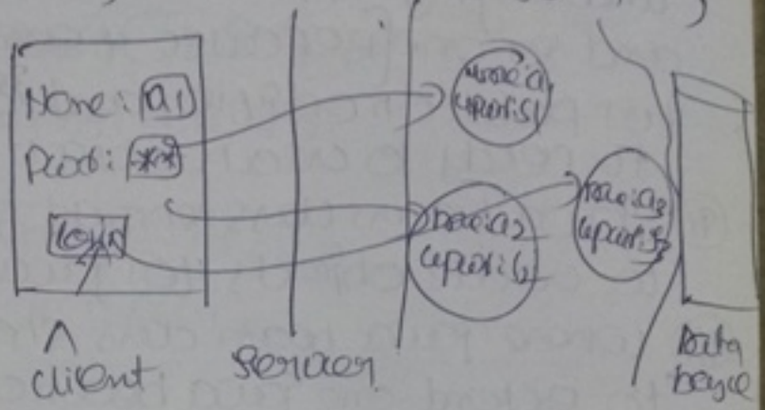
① <jsp:useBean ---->
② <jsp:setproperty ---->
③ <jsp:getproperty ---->
④ <jsp:include ---->
⑤ <jsp:forward ---->
⑥ <jsp:param ---->
⑦ <jsp:plugin
⑧ <jsp:fallback ---->
⑨ <jsp:params ---->
⑩ <jsp:declaration ---->
⑪ <jsp:scriptlet ---->
⑫ <jsp:expression ---->

<jsp:useBean ----> tag :-

Java Beans :- Java bean is a reusable component, a normal java class to represent a particular user form at server side.

The main purpose of java bean component in web applications is to store a particular user form instance at server side.

— The main purpose to store user form instance at server side is to perform the database operations like insert, retrieve and so on--, to perform data validations at server side, before using the data in application logic



To transfer the data from controller layer to view layer in MVC based web applications (DTO) and so on--

— To prepare Java Bean classes in web application we have to use the following rules and regulations.

① Java Bean class is a normal Java class, which includes setter methods and getter methods with respect to the properties.

② In Java technology, always it is suggestible to implements Serializable interface in Java Bean classes. [∵ because to transfer the bean object from controller layer to view layer]

③ Java Bean class is a normal Java class, it should be public, non abstract and non final.

(i) where java bean class should be public because to bring the scope of java Bean class to the underlying application server (or) framework and so on. (Because frame work need to search for public modifier and zero arg constructor whenever it ready to create object for java bean class)

(ii) java bean class should not be abstract because to create objects for java bean.

(iii) where java bean class should not be final because to extend one java bean class to another java bean class in order to optimize the common properties declarations.

(iv) In java Bean class always it is suggestible to declare all the properties as private and all the below fors as public.

(v) In java bean classes, if we want to provide any constructor then that constructor should be public and zero argument because at the time of creating bean object the underlying server or frame work will search and execute public and zero argument constructor.

example for java bean:

```
public class UserBean Implements
                    java.io.Serializable
{
    private String uname;
    private String epwd;
public void setUname(String uname)
    { this.uname=uname;
    }
```

```
public String getUname()
    { return uname;
    }
public void setUpwd(String upwd)
    { this.upwd = upwd;
    }
public String getUpwd()
    { return upwd;
    }
}
```

2→ The main purpose of <jsp:useBean> tag is to interact with java bean object in order to set the data and to get the data.

The java bean was called as UseBean in jsp, in servlet frame work called as formbean, in jsf frame work called as Managed bean, in hibernate called as persistence bean or pojo class.

Syntax:

```
<jsp:useBean id="_" class="_" type="_"
                                scope="_" />
```

</jsp:useBean>

where id attribute can be used to provide a variable to store bean object reference.
- where class attribute will take the fully qualified name of bean class.
- where type attribute will take a class name refered as bean type.

where scope attribute will take a particular scope to hold bean object reference.

— always it is suggestible to provide either session or application scope to the scope attribute

Ex:8

```
<jsp:useBean id="e" class="employee"
    type="employee" scope="session" />
```

Internal flow

When container encounters jsp:useBean tag then container will pick up class attribute value that is the fully qualified name of bean class, container will perform bean class loading, and instantiation.

After creating the bean class object, the generated reference value will be stored in a variable specified as value to id attribute.

After getting bean reference in the form of a variable container will store that bean object reference in the respective scope specified as value to scope attribute.

The above line is almost all equal to

```
Employee e = new Employee
    session.setAttribute("e", e);
```

type of the variable

Date: 22nd May 2012 (Tue)

<jsp:setProperty-->

In jsp technology by using <jsp:useBean--> tag we are able to create a particular bean class object and we are able to maintain bean object reference in the form of a particular variable.

- In jsp technology the main purpose of <jsp:setproperty--> tag is to execute a particular setter method in order to set a value to bean object.
- In jsp technology, we are able to use <jsp:setproperty--> tag as a child tag to <jsp:useBean--> tag.

<jsp:setproperty name="__" property="__" value="__"/>

where name attribute will take a variable which is already maintained bean object reference

Note: The name attribute value in <jsp:setproperty- must be same <id> attribute value -> tag in <jsp:useBean--> tag.

- where property attribute will take the property name to access the respective setter method.
- where value attribute will take a value to pass as parameter to the respective setter method.

ex: <jsp:property name="e" property="eno" value="111" />

The above code is equivalent to

```
Employee e = new Employee();
e.getEno(111)
```

<jsp:getproperty→>

The main purpose of <jsp:getproperty→> tag is to execute a particular getter() method with respect to a property in order to get its value.

In JSP pages, this tag must be utilized as child tag to <jsp:useBean→> tag.

Syntax:

<jsp:getproperty name="_" property="_"/>

— where name attribute will take the variable which we maintained the respective bean object reference.

Note: name attribute value in <jsp:getproperty→> tag must be same as the id attribute value specified as parameter to in <jsp:useBean→> tag

— where property attribute will take the property name to execute the respective getter method.

Ex: <jsp:getproperty name="e" property="eno"/>

2) The above code is equivalent to

```
Employee e = new employee()
e.getEno();
```

Ex8

usebeanapp
- productdetails.html
- productdisplay.jsp
- WEB-INF
  - classes
  - formdss.productclass

productdetails.html

```
<html>
<head> <b> <center> <font size="7" color="red">
    Product Details form.
  </font> </b> </center> </head>
<br> <hr> <br>
<body bgcolor="lightblue">
  <b> <font size="7"> <br>
  <form method="get" action="./productdisplay.jsp">
<pre>
productId <input type="text" name="pid"/>
product Name <input type="text" name="pname"/>
product cost <input type="text" name="pcost"/>
  <input type="submit" value="Display"/>
</pre> </form> </font> </b> </body> </html>
```

product.java

```
package com.dss;
import java.io.*;
public class product implements Serializable
{
    private String pid;
    private String pname;
    private int pcost;
public void setpid(String pid)
    { this.pid = pid;
    }
public String getpid()
    { return pid;
    }
public void setpname(String pname)
    { this.pname=pname;
    }
public String getpname()
    { return pname;
    }
public void setpcost(int pcost)
    { this.pcost=pcost;
    }
public int getpcost()
    { return pcost;
    }
}
```

> javac -d *.java

productDisplay.jsp

```
<! String pid, pname;
   int pcost;
  %>
<%
  try
   {
     pid = request.getParameter("pid");
    pname = request.getParameter("pname");
pcost = Integer.parseInt(request.getParameter
                                ("pcost"));
 }
 catch(Exception e)
    {  e.printStackTrace();
     }
 %>
<html>
  <body bgcolor="dfgkjycolor">
  <b><font size="7"><br><br>
 <jsp:useBean id="p" class="com.dgg.product"
  type="com.dgg.product" scope="sessionScope" >
<jsp:setProperty name="p" property="pid"
 value='<%=pid%>' />
          use single quotations.
<jsp:setProperty name="p" property="pname"
   value='<%=pname%>' />
<jsp:setProperty name="p" property="pcost"
   value='<%=pcost%>' />
```

productId -- <jsp:getproperty name="p"
property="pid"/>

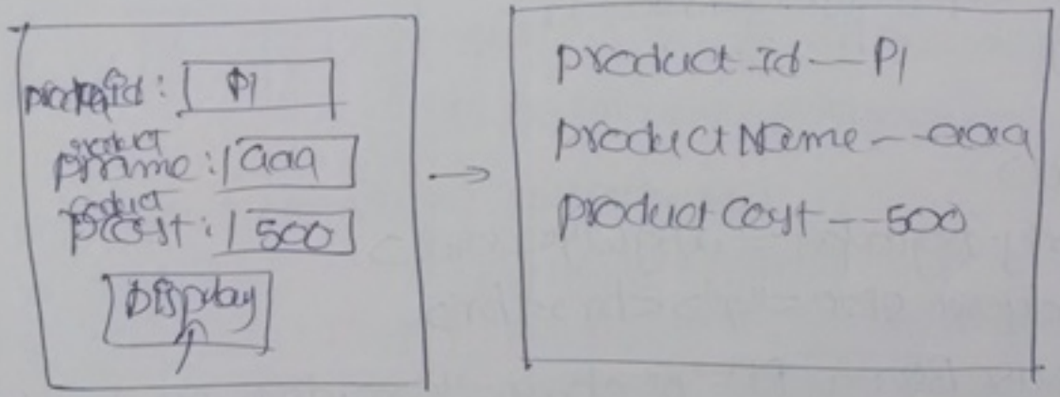
productName -- <jsp:getproperty name="p"
property="pname"/>

productCost -- <jsp:getproperty name="p"
property="pcost"/>

</jsp:useBean>

  </body> </html>

2) Client Server
open
type.



<jsp:include --> tag

Int
Question: what are the differences b/w include directive and include action tag?

Ans: ① In jsp technology include directive can be used to include the target resource content into the present jsp page content.

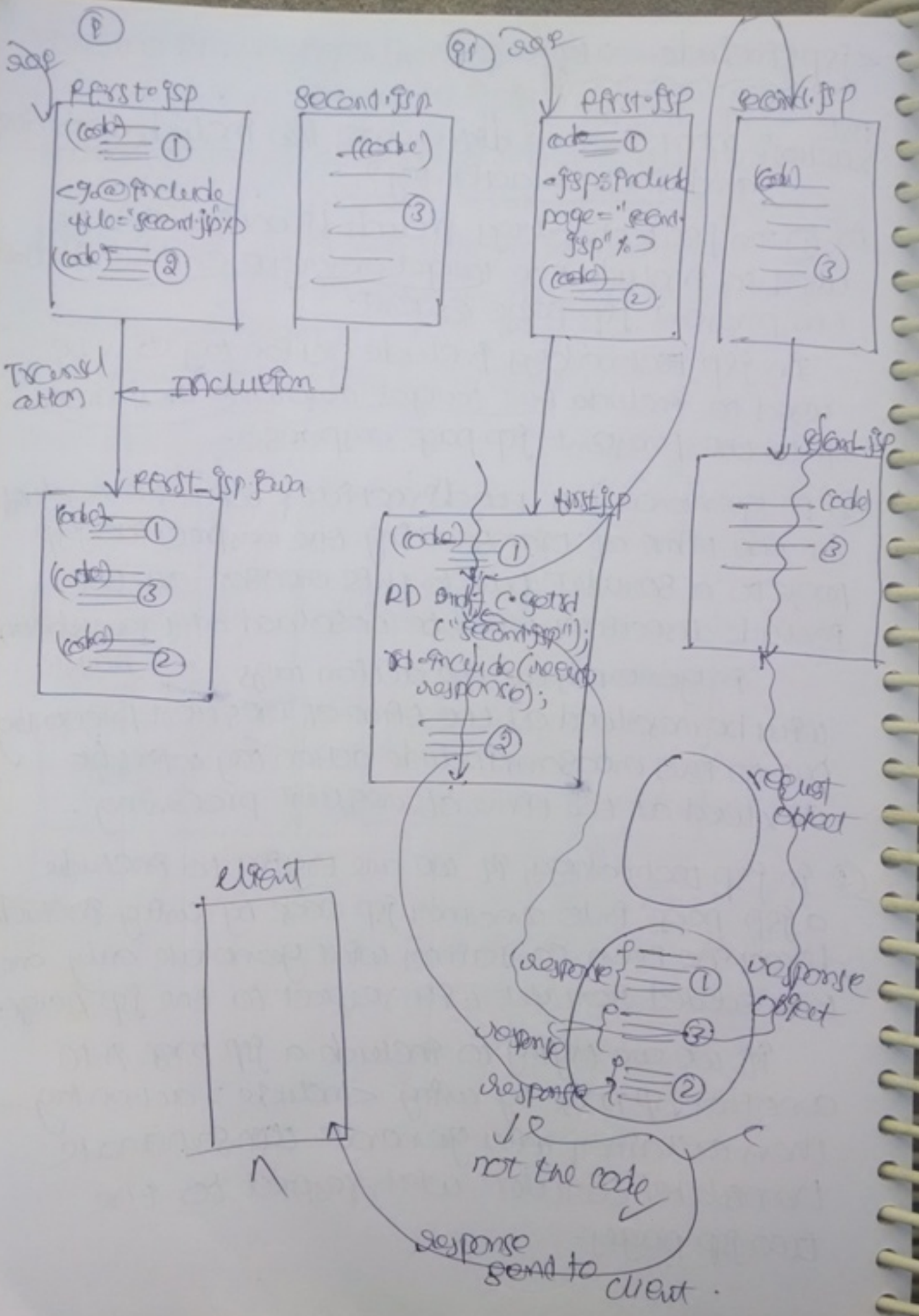
In jsp technology include action tag can be used to include the target resource response into the present jsp page response.

② In general all the directives will be resolved at the time of translating the respective jsp page to a servlet. Due to this reason all the include directives will be resolved at translation phase.

In general, all the action tags will be resolved at the time of request processing. Due to this reason, include action tag will be resolved at the time of request processing.

③ In jsp technology, if we are trying to include a jsp page into another jsp page by using include directive then container will generate only one translated servlet with respect to the jsp page.

If we are trying to include a jsp page into another jsp page by using <include> action tag then container will generate two separate translated servlet with respect to the two jsp pages.



④ Include directive will include the target resource content into the present JSP content at the time of translation, so that include directive will provide static inclusion of data. But include action tag will include the target resource response into the present JSP resource response at the time of request processing, so that include action tag will provide dynamic inclusion.

⑤ In JSP technology include directive can be utilized to include static resources where the frequent modifications are not available. But include action tag can be utilized to include the dynamic resources where the frequent modifications are available.

Date: 23rd may, 2012 Wed

Syntax:

```
<jsp:include page="_" flush="_"/>
```

where page attribute will take the name and location of target resource.

— where flush attribute is a boolean attribute. It can be used to give an information to the container about to autoflush (or) not to autoflush response to the client. JSP writer buffer filled with the response.

Note: This attribute is same as autoflush in page directive.

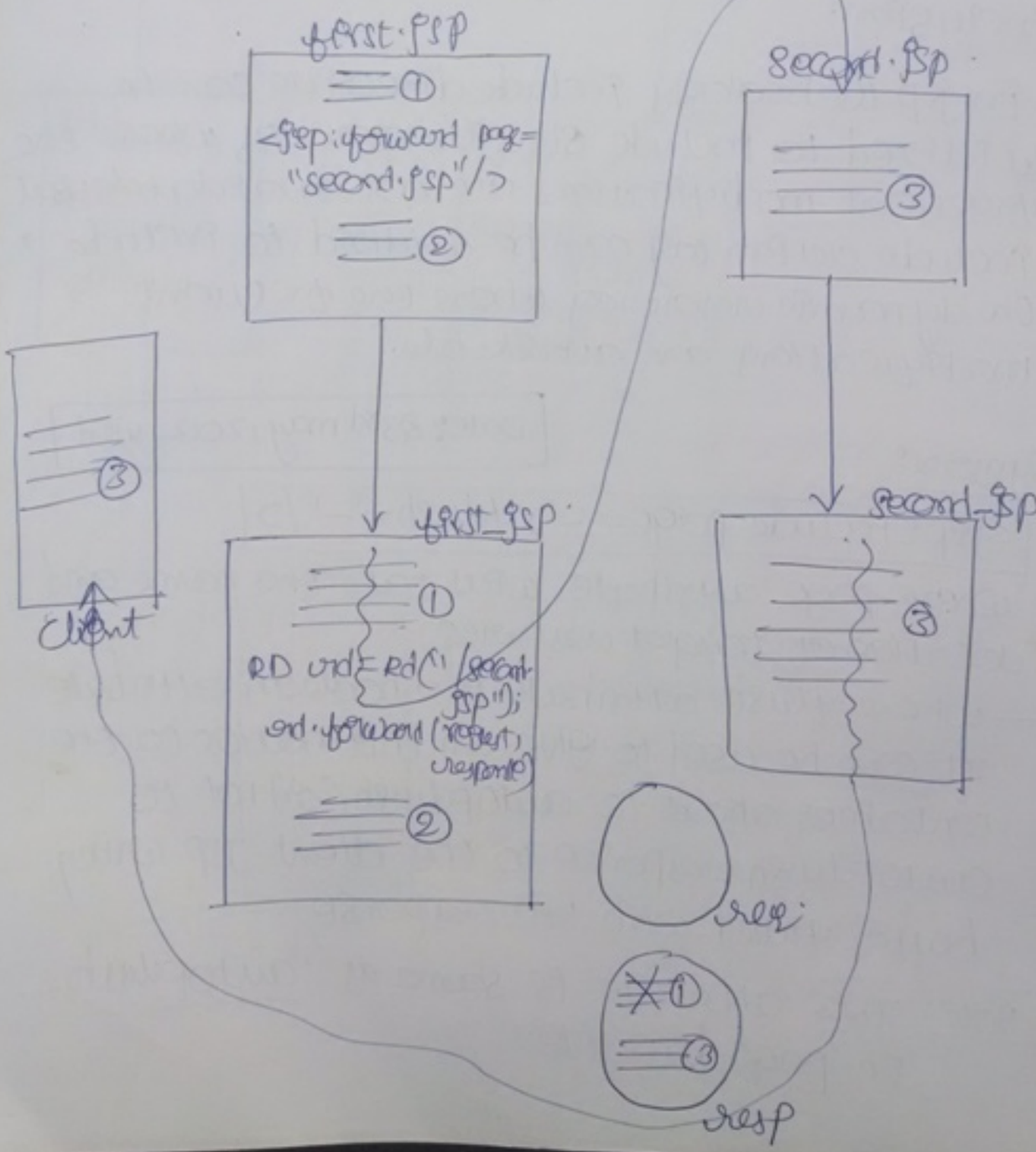
<jsp:forward/>

The main purpose of the <jsp:forward/> tag is to forward request from present jsp page to the target resource

Syntax:

```
<jsp:forward page="—"/>
```

where page attribute will take the name and location of the target resource.



- In case of <jsp:forward-->; when we send a request to first.jsp then a servlet will be created with the name first_jsp, where forward() method will be generated with RequestDispatcher object with respect to the <jsp:forward--> tag.

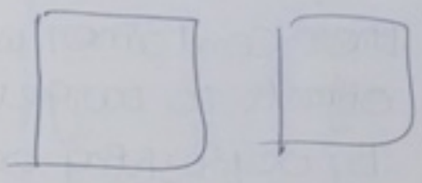
```
RequestDispatcher rd = RequestDispatcher
rd.forward(request, ("second.jsp");
           response);
```

- When container encounter forward() method, then container will bypass request and response objects to target resource that is second.jsp by refreshing response object that is by eliminating previous response from response object
- By the execution of the target resource some response will be added to the response object, at the end of the target resource, container will dispatch the complete response to client without moving back to the first resource.
- Therefore in case of jsp:forward client is able to get only target resource response.

<jsp:param --> tag :- This tag can be used to include a new name-value pair in request object while by passing request and response objects from present resource to the target resource as part of including target resource response into the present resource response or forwarding request object from present resource to the target resource.

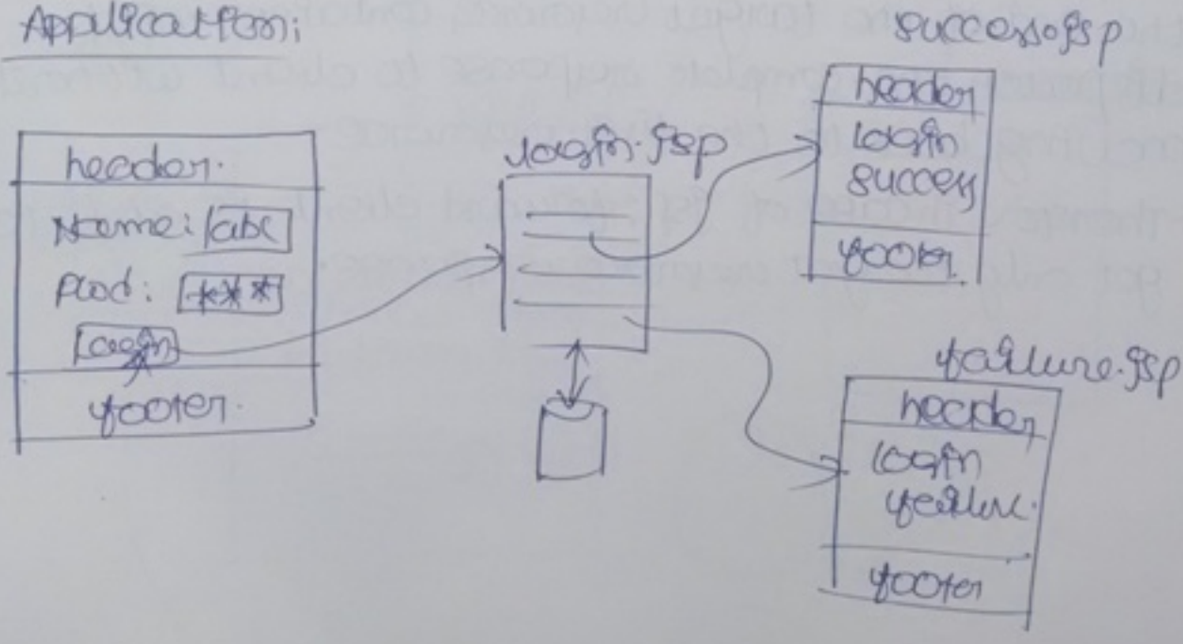
<jsp:param name="__" value="__" />

where name and value attributes can be used to specify name value pair.



— In jsp technology this tag must be used as child tag to <jsp:include> and <jsp:forward> tags.

Application:



loginapp

- loginform.jsp
- header.jsp
- footer.html
- login.jsp
- success.jsp
- failure.jsp
- WEB-INF
  - web.xml
  - lib
    - ojdbc6.jar ✓

header.jsp

```
<%@page import="java.util.*" %>
<html>
  <body>
   <%= new Date() %>
<center> <b> <font size="7" color="red">
  Durga software solutions.
  </font> </b> </center> </body> </html>
```

footer.jsp

```
  <html>
   <body>
    <center> <b> <font size="6">
copy rights 2010 - 2020 @ www.durgasoft.com
  </font> </b> </center> </body> </html>
```

login.jsp (form)

<jsp:include page="header.jsp" flush="true" /> → dynamic resource ✓

```
<html>
 <body bgcolor="lightblue">
 <br><hr><br>
 <b><font size="4">
 <form method="post" action="./login.jsp">
  <pre>
  Name: <input type="text" name="uname" />
 password: <input type="password" name="pwd" />
<input type="submit" value="login" />
   </pre></form></font></b></hr></body>
</html>

<%@include file="footer.html" %>
```

↳ static page.

explanation

login.jsp

```
<%@page import="java.sql.*" %>
<%!
  Connection con;
  Statement st;
  ResultSet rs;

%>
<%
  try
  {
Class.forName("oracle.jdbc.driver.OracleDriver");
con=DriverManager.getConnection("jdbc:oracle:
  thin:@localhost:1521:xe","system","durga");
  st=con.createStatement();
String uname=request.getparameter("uname");
String upwd=request.getparameter("upwd");
ResultSet rs=st.executeQuery("select * from
login where uname='"+uname+"' and upwd=
'"+upwd+"'");
  boolean b=rs.next();
  if(b==true)
  {
%>
<jsp:forward page="success.jsp"/>
```

java only

jsp environment

Java environment using of scriptlet tags

```
<%
}
else
{
%>
```

JSP environment

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

success.jsp

```
<jsp:include page="header.jsp" flush="true" />
<html>
<body bgcolor="lightyellow">
<br><br><br><br>
<b><font size="7" color="green">
Login Success
</font></b><br><br><br><br>
</body></html>
<%@include file="footer.html"%>
```

failure.jsp

```
<jsp:include page="header.jsp" flush="true"/>
<html>
<body bgcolor="lightyellow">
<br><br><br><br>
<b><font size="7" color="red">
Login failure.
</font></b><br><br><br><br>
</body></html>
<%@ include file="footer.html" %>
```

web.xml
to call
our
file

<jsp:declaration ---> tag:-

This action tag is able to represent the scripting element declaration. This action tag can be used to provide all the java declarations in the present jsp page. [There is no difference in the functionality of declarative scripting element and this declaration action tag. This action tag is just an alternative to the declarative scripting element]

Syntax: <jsp:declaration>
java declarations.
<jsp:declaration>

<jsp:scriptlet> </jsp:declaration>

- This action tag can be used to represent the scripting element scriptlet
- This action tag can be used to provide a block of java code inside the jsp page.

syntax: <jsp:scriptlet>
java code.
</jsp:scriptlet>

<jsp:expression --> :-

- This action tag can be used to represent the scripting element expression.
- This action tag can be used to evaluate a single java expression and display the results on client browser.

Syntax: <jsp:expression>
==
<|jsp:expression>

expl
pgm : <%@page import="java.util.*" %>
<jsp:declaration>
Date d=null;
</jsp:declaration>
<jsp:scriptlet>
d=new Date();
String date=d.toString();
</jsp:scriptlet>
<html>
<body bgcolor="lightyellow">

Date --- <jsp:expression> date </jsp:expression>
 </body></html>

<jsp:plugin --> tag:

- The main purpose of <jsp:plugin --> tag is to embed or to include an applet into the present JSP page.

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

where code attribute will take the fully qualified name of the respective applet class.

where width and height attributes we can be used to provide width & height of the applet which are displaying on web page.

- when JSP container encounter <jsp:plugin> tag then container will pickup code attribute value that is the respective applet.class file name then perform applet life cycle, applet loading, applet instantiation, applet initialization, applet start, applet stop and applet destroy.
- In the above context, when JSP container is unable to load or initialize the specified applet then to display an alternative message, we will use <jsp:fallback> tag

Syntax:

<jsp:fallback>
(description)
</jsp:fallback>

This fallback is called.

- In the above context, if we want to pass any initialization parameter to the respective applet then we have to use <jsp:param> tag

Syntax: `<jsp:param name="_" value="_" />`

- If we want to pass multiple number of initialization parameters to the respective applet then we have to use <jsp:param> tags in multiple number of times but all the <jsp:param> tags must be enclosed with <jsp:params> tag.

Syntax:
```
<jsp:params>
    <jsp:param name="_" value="_" />
    ==========
</jsp:params>
```

pluginapp
- logo.jsp
- LogoApplet.class
- WEB-INF
  - classes

LogoApplet.java

```
import java.awt.*;
import java.event.*;
import java.applet.*;
public class LogoApplet extends Applet
    { public void paint(Graphics g)
        {
```

```
Font f = new Font("aerial", FONT.BOLD, 30);
        g.setFont(f);
    this.setForeground(Color.red);
    this.setBackground(Color.yellow);
      g.drawString("Durga", 100, 200);
    }
}
```

logo.jsp

```
<jsp:plugin code="LogoApplet" width="500"
 height="500"
 type="applet">
 </jsp:plugin>
```

— Compilation and execution

Custom Actions :-

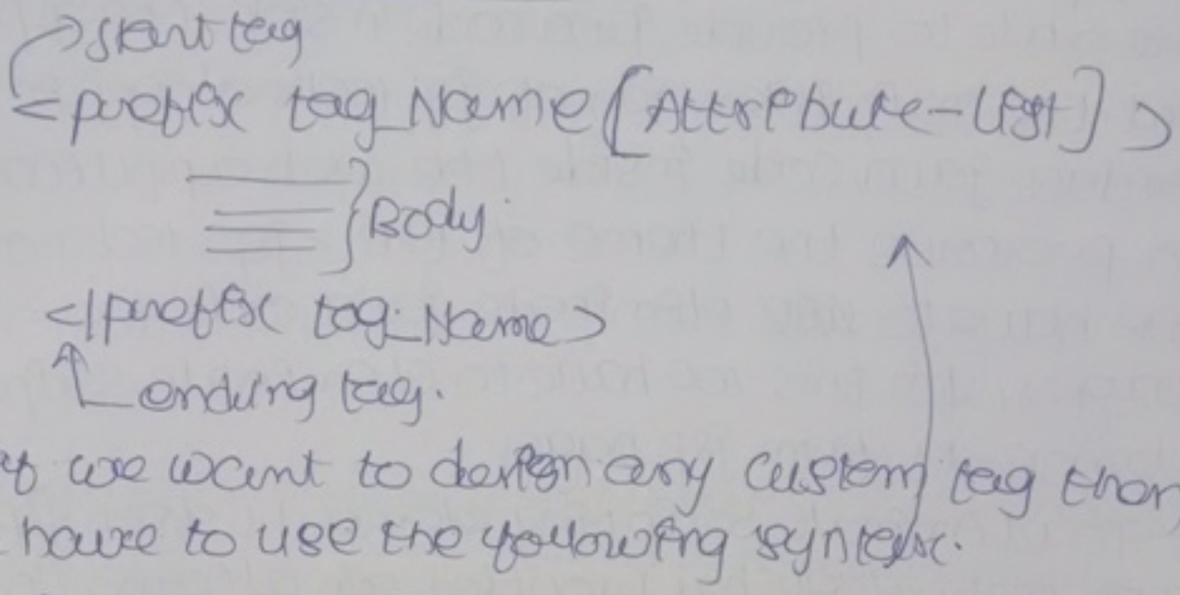
- In jsp technology by using scripting elements we are able to provide java code inside the jsp page but the main intention of jsp technology is to reduce java code inside the web application.
To preserve the theme of the <jsp:technology we have to eliminate java code from jsp pages, for this we have to eliminate scripting elements from jsp pages.
- To eliminate scripting elements from jsp page Jsp technology has provided an alternative in the form of jsp actions. In jsp actions we have already standard actions but which are in limited number and having bounded functionality.

Due to this limited number of standard actions and bounded functionality, standard actions are not sufficient to eliminate java code completely from jsp pages.

- In the above context, to eliminate java code completely from jsp page we have to use an alternative in the form of custom actions.
- Custom actions are the user defined actions, which would be prepared by the developer as per their application requirements.
- In general in web applications, we can represent all the custom actions by defining a set of user defined tags called as

Custom tags:

Syntax for custom tags:-



- If we want to define any custom tag then we have to use the following syntax.

- In JSP applications, if we want to define any custom tag then we should require the following three elements.
  1. taglib directive in a JSP page.
  2. a tld file. [taglibrary descriptor tag]
  3. taghandler class.

Date: 25th may, 2012 Fri

The main purpose of the taghandler class is to define a block of java code inorder to implement the basic functionality to a particular custom tag.

— The main purpose of tld file is to provide the mapping between the respective custom tag name and the respective tag handler class.

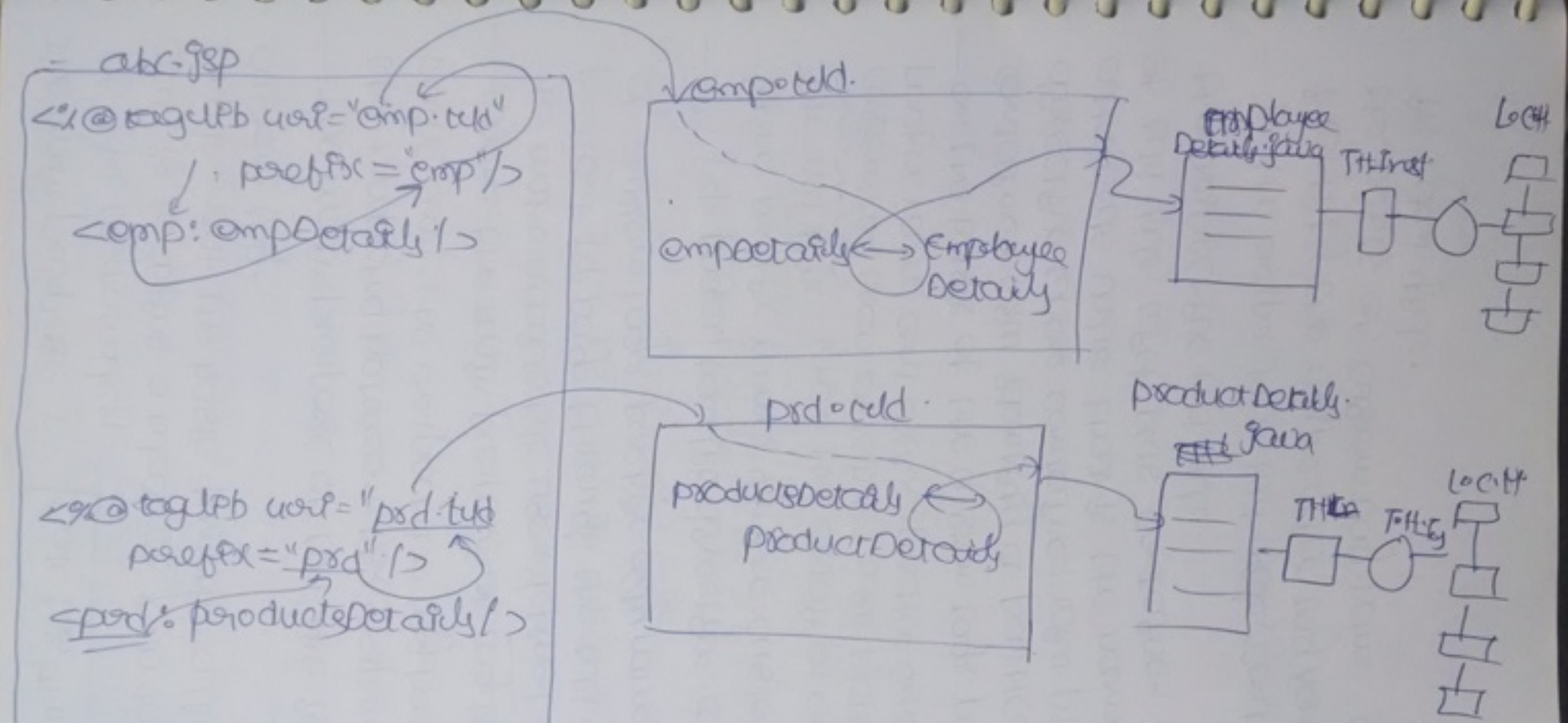
— where the main purpose of taglib directive is to make available user-defined library to the present jsp page through the tld file on the basis of custom tags prefix names.

— In jsp pages, taglib directive can be used to define prefix names for the custom tags.

Internal flow: when the container encounters a particular custom tag then container will pickup custom tag prefix name and the respective custom tag name.

— on the basis of the custom tags prefix name container will identify a particular taglib directive where container will pickup uri attribute value that is the name and location of tld file, with this container will identify the respective tld file.

— In the respective tld file, container will identify the respective tag handler class on the basis of custom tag name and identify its .class file.



— When container recognize the required tag handler class .class file then container will perform tag handler class loading, instantiation and execute all the life cycle methods.
— By the execution of the respective tag handler class only container will perform the required action with respect to the custom tags.

tld file :-

The main purpose of the tld file in custom tag design is to provide the mapping between custom tags name and the respective tag handler class and it able to provide metadata or description about custom tags attributes.
— In web applications all the tld files must be saved with .tld extension. In general in web applications all the tld files will be recognized and parsed at the time of server startup by the container.
— In web applications, it is possible to provide the tld files at any location of the web application directory structure.
— In web applications it is possible to provide more than one tld file but it is depending on the types of custom tags which are prepared.

- If we want to provide the mapping between custom tag name and the respective tag handler class then we have to use the following tags.

```
<taglib>
  <tlib-version> tld file version no </tlib-version>
  <jsp-version> jsp1.1 or above </jsp-version>
  <short-name> tld file short name </short-name>          (optional)
  <description> tld file description </description>
  <tag>
      <name> custom tag name </name>
      <tag-class> fully qualified name of custom tag handler class. </tag-class>
      <body-content> jsp/empty </body-content>
      <short-name> short name of the custom tag </short-name>          (optional)
      <description> description about custom tag </description>
  </tag>
  ---
</taglib>
```

TagHandler class:

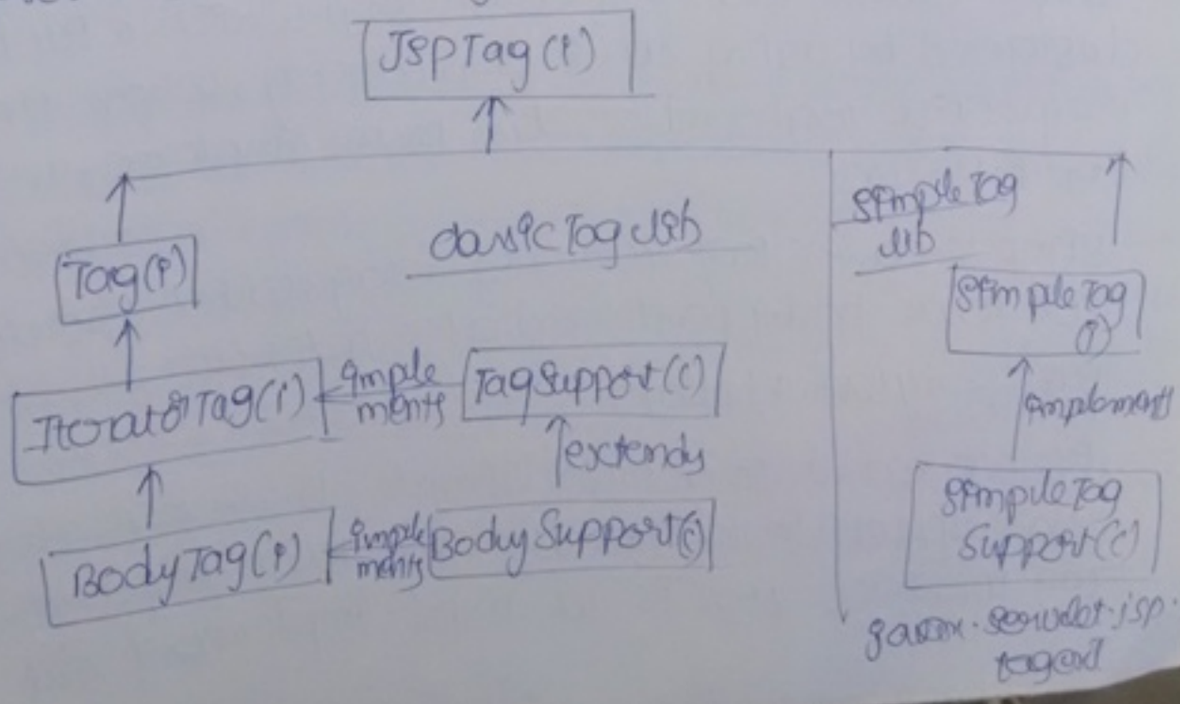
The main purpose of the taghandler class in custom tag design is to define the basic definition for the respective custom tag.

- Taghandler class is a normal java class, which should not require predefined library provided by jsp technology to implement custom tags functionality.
- To design taghandler classes jsp technology has provided a complete predefined library in the form of the following packages.

javax.servlet.jsp.
javax.servlet.jsp.tagext

Date: 26th may, 2012 sat

- To design taghandler classes in custom tags javax.servlet.jsp.tagext has provided the following predefined library.



- As per the predefined library provided by JSP technology, there are two types of custom tags.
  1. classic tags
  2. simple tags.

① classic tags :-

classic tags are the custom tags, which will be designed by using classic tag library.

- In case of classic tag library if we want to design custom tags then the respective tag handler class must implement Tag interface either directly or indirectly
- As per the classic tag library provided by JSP technology there are three types of custom tags.
  ① simple classic tags. classic
  ② Iteration tags
  ③ body tags.

① simple classic tags :-

- simple classic tag is a classic tag, which will be designed by using classic tag library, where the respective tag handler class must implement tag (Interface).
- simple classic tag is a custom tag, which should not have body part and attributes list.
  ex: <mytags:hello/>
- If we want to design simple classic tags, where the respective tag handler class must implement tag interface that is we must implement all

the methods which are available in Tag interface

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

public class MyHandler implements Tag
    {
    }
```

— In JSP applications when container encounter a custom tag then container will access setPageContext() method in order to inject pageContext object.

— After the setPageContext() method container will access setParent() method. If the present tag is really a child tag to a particular parent tag then container will access setParent() method by passing parent tag taghandler class object

reference as parameter otherwise container will access setparent() method by passing itself as a value.

— After executing setparent() method container will access getparent() method inorder to get the parent tags taghandler class object, if the present tag is child tag otherwise container will skip getparent() method execution.

— To perform a particular action with respect to start tag of the custom tag container will access doStartTag() method.

— After executing doStartTag() method executing the custom tag body or not is completely depending on the return value which we are going to return from doStartTag() method.

— there are two possible return values from doStartTag() method.

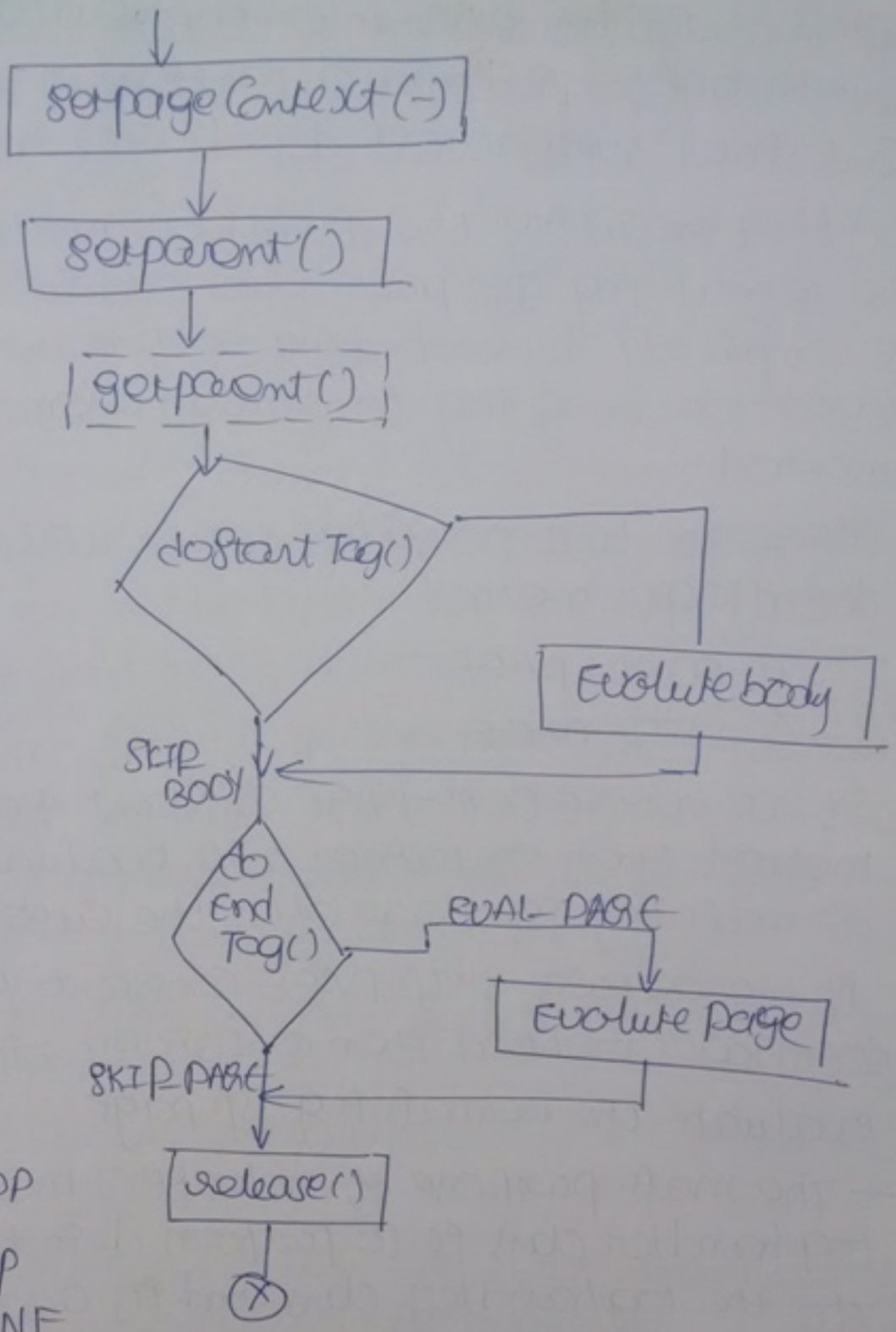
① EVAL_BODY_INCLUDE

② SKIP_BODY.

— If we return EVAL_BODY_INCLUDE from doStartTag() method then container will evaluate custom tag body.

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

- When container encounter the end tag of the custom tag to perform a particular action container will access doEndTag() method.
- After executing the doEndTag() method evaluating the remaining JSP page after the custom tag is completely depending on the return value which we are going to return from doEndTag() method.
- There are two possible return values from doEndTag() method
  1. EVAL_PAGE
  2. SKIP_PAGE.
- If we return EVAL_PAGE constant from doEndTag() method then container will evaluate the remaining JSP page after the custom tag.
- If we return SKIP_PAGE constant from doEndTag() method then container will not evaluate the remaining JSP page.
- The main purpose of release() method in tag handler class is to perform deinstantiation for the tag handler class and its associated objects.



helloTagApp
- hello.jsp
- WEB-INF
  - hello.tld
  - classes
    - com.dss.Hello.class

hello.jsp

```
<%@taglib uri="/WEB-INF/hello.tld"
                                prefix="mytags"%>
<html>
<body bgcolor="lightyellow">
<center><b><font size="7" color="red">
<br><br>
<mytags:hello/>
</font></b></center></body></html>
```

hello.tld

```
<taglib>
 <tlib-version>1.0</tlib-version>
 <jsp-version>2.0</jsp-version>
 <tag>
  <name>hello</name>
  <tag-class>com.dss.Hello</tag-class>
  <body-content>empty</body-content>
 </tag>
</taglib>
```

Hello.java

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
  public class Hello implements Tag
    { public pageContext pageContext;
      public Tag t;
```

```
public void setPageContext(PageContext pageContext)
    { this.pageContext = pageContext;
    }
public void setParent(Tag t)
    { this.t = t;
    }
public Tag getParent()
    { return t;
    }
public int doStartTag() throws JspException
  { try
    {
JspWriter out = pageContext.getOut();
    out.println("Hello User");
    }
    catch(Exception e)
        { e.printStackTrace();
        }
    return SKIP_BODY;
  }
public int doEndTag() throws JspException
        { return SKIP_PAGE;
        }
public void release()
        { }
    }
```

Date: 27th may, 2012 Sun

— To compile the above tag handler class, we must set class path environment variable to jsp_api.jar file, which is available in tomcat server at the following location.

C:\Tomcat 7.0\lib\jsp-api.jar

higher

— In the custom tags design if we provide body content type empty, but if you provide body to the custom tag in jsp page then the container will raise an exception like

eg. apache.jasper.JasperException: /hello.jsp(6,0)
According to tld, tag my tags: hello must be empty but it not

attributes in custom tags

If we want to provide attributes in custom tags then we have to provide the following 3 steps.

① define custom tag attribute in jsp page.

eg: <mytags:hello name="durga"/>

② provide the description of attribute in tld file.
If we want to define any attribute in custom tag then we have to configure it in the respective tld file, for this we have to use the following tags:

<taglib>
<tag>
<attribute>
<name> attribute name </name>
<required> true/false </required>
<rtexprvalue> true/false </rtexprvalue>
</attribute>
</tag>
</taglib>

— where <attribute> tag can be

where <name> tag will take the attribute name which we defined in custom tag. where <required> tag will take boolean value and it will make the attribute as mandatory or optional attribute.

— where <rtexprvalue> tag will make the respective attribute to accept or not runtime evaluated exp

Step ③: Declare a property on the respective getter method with the same name of the attribute in the respective tag handler class.

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

Iteration tags:

- Iteration tag is a classic tag, it will allow to perform number of evaluations on custom tag body
- If we want to design iteration tags then the respective taghandler class must implement IterationTag interface either directly or indirectly
- In case of the Iteration tags, at taghandler class doStartTag() method must return EVAL_BODY_INCLUDE constant

```
public interface IterationTag extends Tag
{
public static final int EVAL_BODY_INCLUDE;
                        SKIP_BODY;
                        EVAL_PAGE;
                        SKIP_PAGE;
                        EVAL_BODY_AGAIN;
```

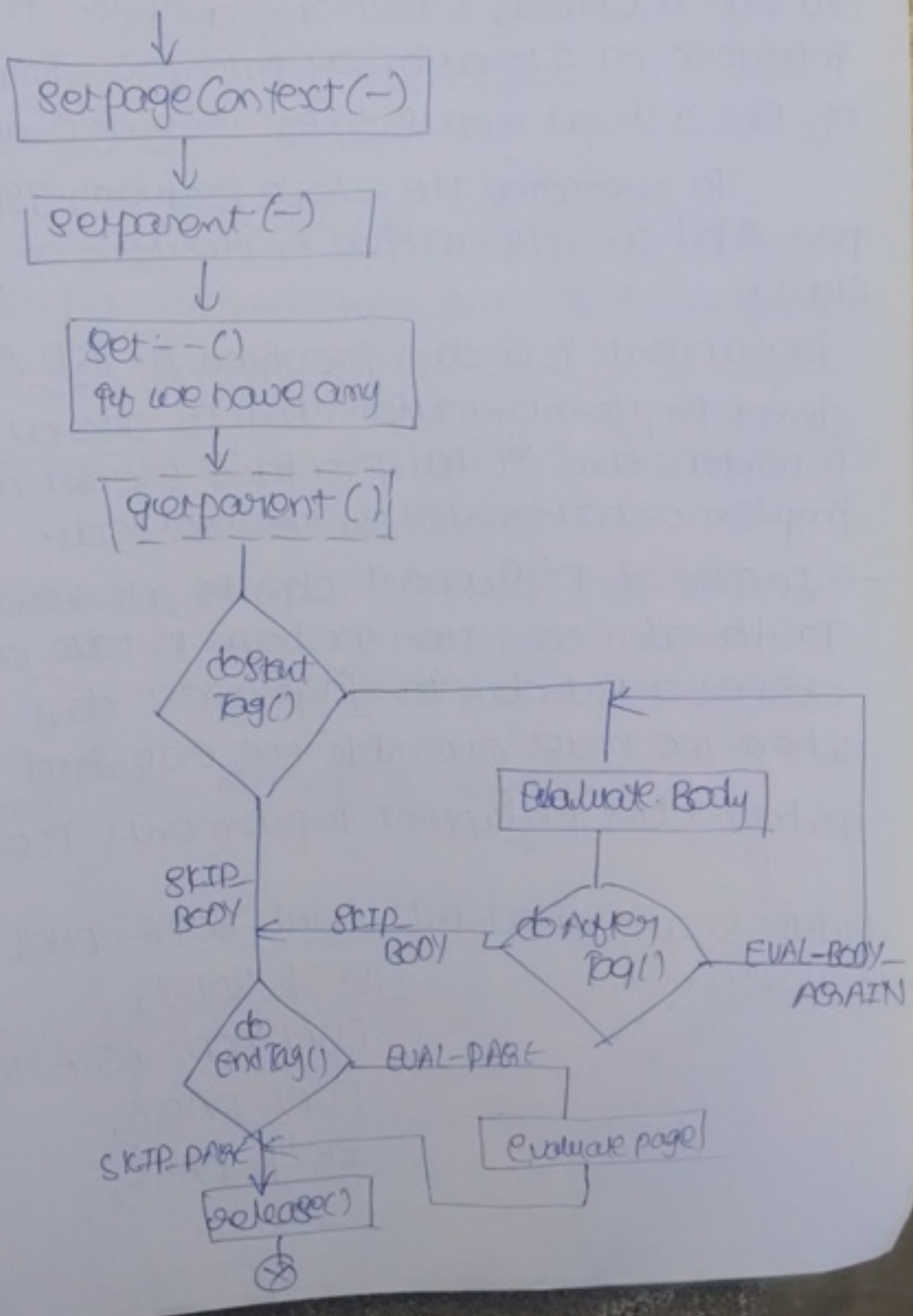
public void setPageContext(PageContext pageContext);
public void setParent(Tag t);
public Tag getParent();
public int doStartTag() throws JspException;
public int doAfterBody() throws JspException;
public int doEndTag() throws JspException;
public void release()
}
public class MyHandler implements IterationTag
{
}

— In case of IterationTags, we must return EVAL_BODY_INCLUDE from doStartTag() method. If we do like this container will evaluate the custom tag body, at the end of the custom tag body container will access doAfterBody() method. In the above context evaluating the custom tag body again or terminating the custom tag body evaluation is completely depending on the return value which we will return from doAfterBody() method.

— There are two possible return values from doAfterBody() method

① EVAL_BODY_AGAIN
② SKIP_BODY

— If we return EVAL_BODY_AGAIN from doAfterBody() method then container will evaluate the custom tag body again. If we return SKIP_BODY constant from doAfterBody() method then container will terminate the custom tag body evaluation.



— Upto now, in the custom tags design if we want to design any custom tag then the respective taghandler class must implement either Tag interface or IterationTag interface.

— If we implement either of these interfaces then we must provide the implementation for all the methods which are declared in the Tag interface or IterationTag interface irrespective of the actual application requirement.

To overcome the above problem, JSP api has provided an alternative in the form of TagSupport class.

TagSupport is a class provided by JSP API as direct implementation class to IterationTag interface and it has provided the default implementation for all the methods.

— In case of TagSupport class if we want to design TagHandler class then we have to take an user class as a subclass to TagSupport class defined where we must override the required method.

```
public class TagSupport implements IterationTag
{
public static final int EVAL_BODY_INCLUDE;
                        SKIP_BODY;
                        EVAL_BODY_AGAIN;
                        EVAL_PAGE;
                        SKIP_PAGE;
```

```
Public pageContext pageContext;
    Public Tag t;
Public void setPageContext(pageContext context)
    { this.pageContext = pageContext;
    }
Public void setParent(Tag t)
        { this.t = t;
        }
Public Tag getParent()
        { return t;
        }
Public int doStartTag() throws JspException
        { return SKIP_BODY;
        }
Public int doAfterBody() throws JSPException
        { return SKIP_BODY;
        }
public int doEndTag() throws JSPException
        {
        return EVAL_PAGE;
        }
 public void [illegible] release()
            { }

    }
public class MyHandler extends TagSupport
        { =
        }
```

Iterationtagapp:

- Iterate.jsp
- WEB-INF
  - Iterate.tld
  - classes
    - com.dss.Iteration.class

Iterate.jsp

```
<%@taglib url="/WEB-INF/Iterate.tld" prefix="mytags"
                                                %>
<html>
 <body bgcolor="lightgreen">
 <center><b><font size="4">
  <br><br>
  <mytags:Iterate time="10">
    Durga Software Solutions<br>
    </mytags:Iterate>
 </font></b></center></body></html>
```

Iterate.tld:

```
<taglib>
 <tlib-version>1.0</tlib-version>
 <jsp-version>2.0</jsp-version>
 <tag>
 <name>Iterate</name>
 <tag-class>com.dss.Iteration</tag-class>
```

<body-content>jsp</body-content>
<attribute>
<name>times</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
</taglib>

Iteration.java

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class Iteration extends TagSupport
    { int count=1;
      private int times;
      public void setTimes(int time)
          { this.times=times;
          }
      public int doStartTag() throws JspException
          { return EVAL_BODY_INCLUDE;
          }
      public int doAfterBody() throws JspException
          { if(count<times)
              { count=count+1;
                return EVAL_BODY_AGAIN;
              }
```

```
else
{
return Stop_Body;
}
}
}
```

Nested tags : Declaring a tag inside another tag is called as nested tags.

— In jsp technology we have to define a seperate taghandler class for each and every nested tag and outer tag.

— If we want to get the parent tags taghandler class object reference in the child tags handlerclass then we have to use getparent() method.

nestedtagapp
- nested.jsp
- WEB-INF
  - nested.tld
  - classes
    - com.dss.If.class
    - com.dss.True.class
    - com.dss.False.class

nested.jsp

```
<%@ taglib uri="/WEB-INF/nested.tld" prefix="mytags" %>
<html>
<body bgcolor="dfghtyellow">
<center><b><font size="4" color="red">
<br><br>
<mytags:If condition='<%=10<20%>'>
    <mytags:True>
      condition is true.
    </mytags:True>
```

<mytags:false>
condition is false
</mytags:false>
</mytags:pf>
 </body></html>

nested.tld

<tag-lib>
<tld-version>1.0</tld-version>
<jsp-version>2.0</jsp-version>
<tag>
<name>pf</name>
<tag-class>com.dss.pf</tag-class>
<body-content>jsp</body-content>
<attribute>
<name>condition</name>
<required>true</required>
<rtexprvalue>true</rt
</attribute>
</tag>
<tag>
<name>true
<tag-class>com.dss.True
<body-content>JSP.
</tag>
<tag>
<name>false

```
<tag-class> com.dss.false </tag-class>
<body-content> jsp </body-content>
</tag>
</tagLib>
```

If.java

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

True.java

```
package com.dss;
import javax.servlet.jsp.*;
                .tagext.*;
public class True extends TagSupport
    {
```

```
Public int doStartTag() throws JspException
    {
      If f=(If) getParent();
      boolean b= f.getConditionOn();
       if(b==true)
        { return EVAL_BODY_INCLUDE;
        }
        else
          { return SKIP_BODY;
          }
      }
}
```

False.java :

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class false extends TagSupport
    { public int doStartTag() throws JspExc
                                     eption
       {
      If f=(If) getParent();
      boolean b=f.getConditionOn();
        if (b==true)
          { return SKIP_BODY;
          }
        else
          {
          return EVAL_BODY_INCLUDE;
          }
      }
}
```

customdbapp
- display.jsp
- WEB-INF
  - product.tld
  - classes
    - ojdbc6.jar
    - com.dss.productDisplay.class

SQL> select * from product;

PID	PNAME	PCOST
P1	aaa	500
P2	bbb	600
P3	ccc	700
P4	ddd	800
P5	eee	900
P6	fff	500

display.jsp

```
<%@ taglib uri="/WEB-INF/product.tld" prefix="pd" %>
<pd:productDetails />
```

product.tld

```
<taglib>
<tld-version>1.0
<jsp-version>2.0
<tag>
<name>productDetails
<tag-class>com.dss.productDisplay
<body-content>empty
</tag> </taglib>
```

productDisplay.java

```
package com.dss;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.sql.*;
public class productDisplay extends TagSupport
    { Connection con;
      Statement st;
      ResultSet rs;
  public int doStartTag() throws JSPException
      {
        try
        {
  JspWriter out = pageContext.getOut();
     rs = st.executeQuery("select * from prodba");
ResultSetMetaData md = rs.getMetaData();
  int count = md.getColumnCount();
     out.println("<html>");
                ("<body bgcolor="JSPGreen">");
                ("<center>");
                <table border='1' bgcolor="light
                                          yellow">");
  out.println("<tr>");
    for(int i=1; i<=count; i++)
      { out.println("<td>");
  out.println(md.getColumnName(i));
                ("</td>");
        }
```

```
out.println("</tr>");
while(rs.next())
{
out.println("<tr>");
for(int i=1; i<=count; i++)
{
out.println("<td>");
out.println(rs.getString(i));
out.println("</td>");
}
out.println("</tr>");
}
out.println("</table></center></body></html>");
}
catch(Exception e)
{
e.printStackTrace();
}
return SKIP_BODY;
}
public productDisplay()
{
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
con=DriverManager.getConnection("jdbc:oracle:
thin:@localhost:1521:xe","system","donga");
st=con.createStatement();
}
catch(Exception e)
{
e.printStackTrace();
}
}
}
```

<Body Tags :- In general by using simple classic tags, Iteration tags, nested tags we are able to display the custom tag body as it is what we provided in jsp page.

- As per the application requirements we want to perform modifications on custom tag body.
- To achieve this requirement we have to use body tags.

In case of body tags if we want to design any custom tag then the respective tag handler class must implement BodyTag interface either directly or indirectly.

```
public interface BodyTag extends IterationTag
{
    public static final int EVAL_BODY_INCLUDE;
                            SKIP_BODY;
                            EVAL_PAGE;
                            SKIP_PAGE;
                            EVAL_BODY_AGAIN;
* public static final int EVAL_BODY_BUFFERED;
public void setPageContext(PageContext pageContext);
    public void setParent(Tag t);
    public Tag getParent();
    public int doStartTag() throws JSPException;
* public void setBodyContent(BodyContent
* public void doInitBody();                bodyContent);
```

public int doAfterBody() throws JSPException
public int doEndTag() throws JSPException
public void release();
}

public class MyHandler implements BodyTag
{
==
}

— In case of the body tags, at the tag handler class doStartTag() tag method should not return SKIP_BODY constant.

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

① EVAL_BODY_INCLUDE
② SKIP_BODY
③ EVAL_BODY_Buffered

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

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

— If we return EVAL_BODY_BUFFERED constant from doStartTag() method then container will execute setBodyContent() method by passing BodyContent object as parameter.

where BodyContent is an object, it will manage the provided custom tag body. To get the custom tag body from BodyContent object we have to use the following method

`public String getString()`

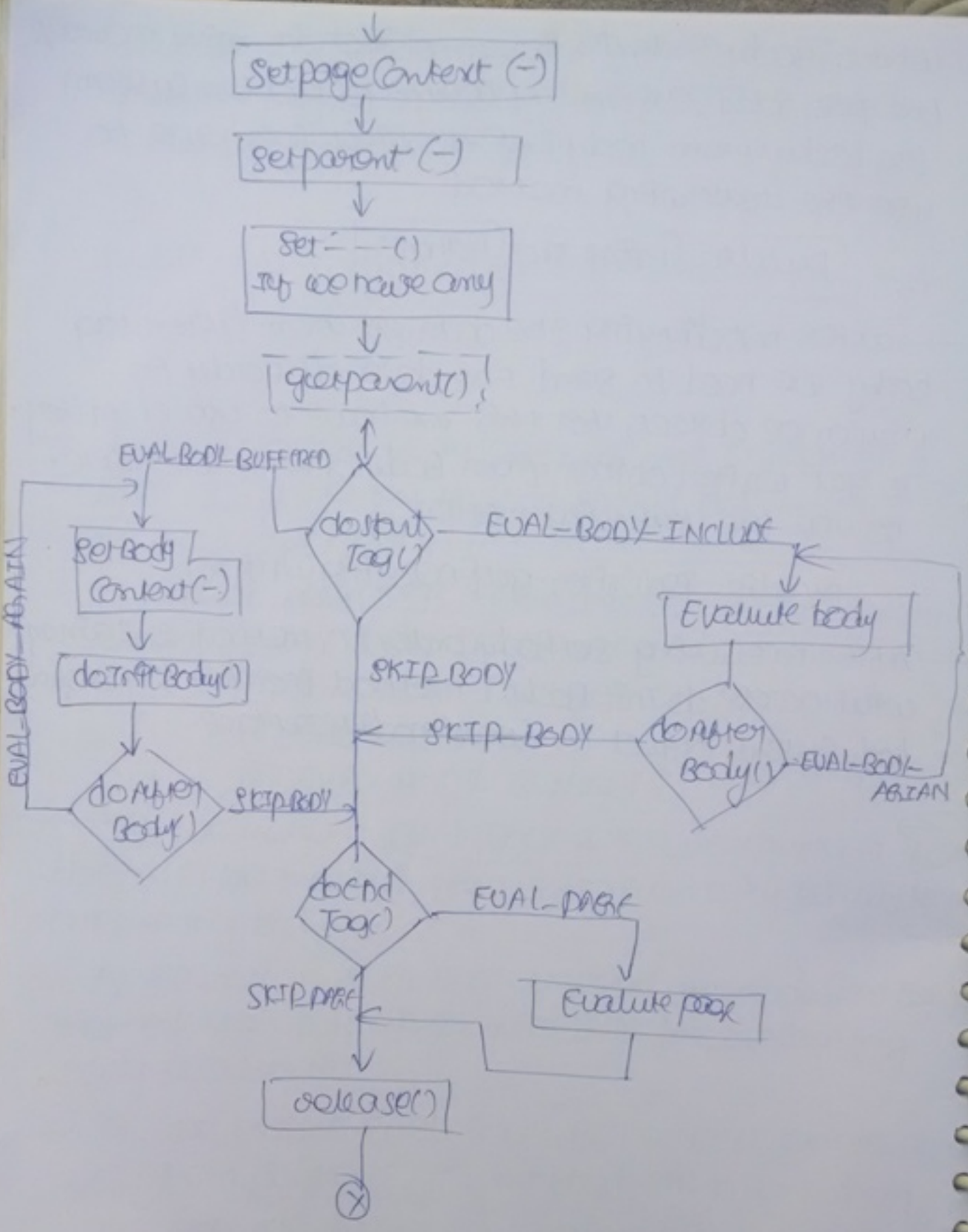
— after modifying the data from custom tag body we need to send the data directly to response object, for this we have to use a writer.

— To get writer object from BodyContent we have to use the following method

`public JspWriter getEnclosingWriter()`

:— After executing setBodyContent() method container will access doInitBody() method inorder to prepare BodyContent object to allow modifications.

⇒



- If we use the above approach to design custom tags then we must implement BodyTag interface above we must provide the implementation for all the methods in BodyTag interface from perspective of the application requirement
- To overcome the above problem, jsp technology has provided an alternative. That is BodyTagSupport class.

If we want to design custom tags, with BodyTag class then we have to take an user defined class as taghandler class, it must be extended from BodyTagSupport class.

```
public class BodyTagSupport extends TagSupport
    {                                implements BodyTag
public static final int EVAL_BODY_INCLUDE;
public static final int SKIP_BODY;
public static final int EVAL_PAGE;
public static final int SKIP_PAGE;
public static final int EVAL_BODY_AGAIN;
public static final int EVAL_BODY_BUFFERED;
    public pageContext pageContext;
      public Tag t;
    public BodyContext bodyContext;
    public void setpageContext(pageContext
      { this.pagecontext = pageContext;  pageContext)
      }
```

```
public void setParent(Tag t)
    { this.t=t;
    }
  public Tag getParent()
      { return t;
      }
public int doStartTag() throws JSPException
    { return EVAL_BODY_BUFFERED;
    }
public void setBody(context (BodyContent
                             bodyContent)
    { this.bodyContent=bodyContent;
    }
  public void doInitBody()
    {}
  public int doAfterBody() throws JSPException
      { return SKIP_BODY;
      }
  public int doEndTag() throws JSPException
      {
      return EVAL_PAGE;
      }
    public void release()
        {
        }
}
public class MyHandley extends TagSupport
    {==
    }
```

bodytagapp

- reverse.jsp

reverse.jsp

```
<%@taglib uri="/WEB-INF/reverse.tld" prefix="mytags"%>
<html>
<body bgcolor="lightyellow">
<center>
<mytags:reverse>
```

reverse.tld

```
<taglib>
 <tlib-version>1.0
 <jsp-version>2.0
  <tag>
    <name>reverse
    <tag-class>com.dss.Reverse
    <body-content>jsp<
    </tag>
    </taglib>
```

Reverse.java

```
package com.dgg;
import javax.servlet.jsp.*;
                .tagext.*;
public class Reverse extends BodyTagSupport
  {
  BodyContent bodyContent;
  public void setBodyContent(BodyContent bodyContent)
    { this.bodyContent = bodyContent;
    }
public int doEndTag() throws JSPException
    {
      try
      {
String body = bodyContent.getString();
StringBuffer sb = new StringBuffer(body);
 StringBuffer rev_Data = sb.reverse();
 JspWriter out = bodyContent.getEnclosingWriter();
    out.println(rev_Data);
     } //try
    catch(Exception e)
      { e.printStackTrace();
      }
    return EVAL_PAGE;
     } //doEndTag
} //class
```

— What are the differences b/w classic tags & simple

Ans ① classic tags are more api dependent tags. but simple tags are less api dependent.

② To design custom tags by using classic tag library then we must remember three different life cycles but to design custom tags by using simple tag library we have to remember only one life cycle.

③ Incase of classic tags, if we want to design any custom tag then the respective tag handler class must implement Tag interface either directly or indirectly.

To design the custom tags by using simple tag library then the respective tag handler class must implement SimpleTag interface either directly or indirectly.

④ Incase of classic tags by default all the custom tags are not body tags. But in case of simple tags by default all the custom tags are body tags

⑤ Incase of classic tags, all the taghandler class objects are cacheable objects, we are able to improve reusability but in case of simple tags all the taghandler class objects are non-cacheable objects.

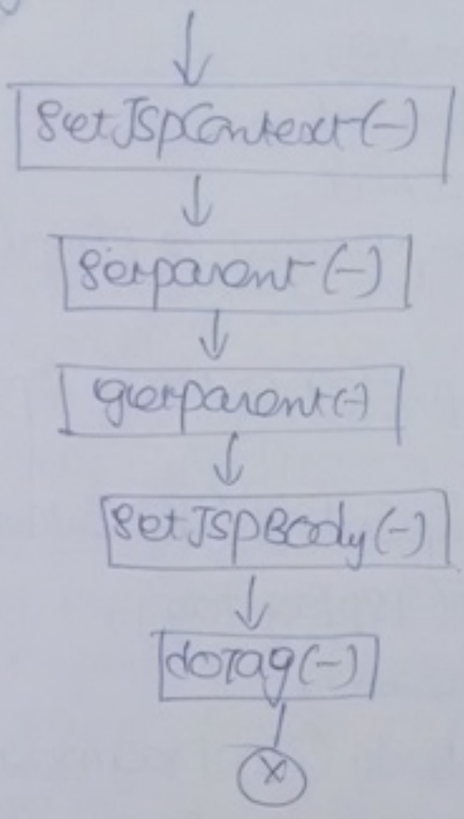
Any the we need to prop object

— If we want to design custom tags by using simple tag library then we should write the respective tag handler class must implement a SimpleTag interface.

```
public interface SimpleTag extends JspTag
    {
      public void setJspContext(JspContext jspcontext);
public void setparent(JspTag tag);
public JspTag getparent();
 public void setJspBody(JspFragment
                                  jspfragment);
public void doTag() throws
                  JspException, IOException
 }
public class MyHandler implements SimpleTag
    { ==
    }
```

— where JspContext is same as pageContext, it can be used to get all the jsp implicit objects.

— where JspFragment is same as BodyContent object, it can be used to accommodate custom tag body.



Note: In case of simple tag library, to handle a custom tag, it is not required to provide doStartTag() method and doEndTag() method, it is sufficient to provide doTag() method.

— To design custom tags, if we use the above approach, then we have to implement all the methods declared in SimpleTag interface.

To overcome the above problem, Jsp technology has provided an alternative, that is SimpleTagSupport class.

— SimpleTagSupport class is direct implementation class to SimpleTag interface, which provides the default implementation for each and every method.

- If we want to design custom tags using SimpleTag support class then the respective taghandler class must be a sub class to SimpleTagSupport class

```
public class SimpleTagSupport implements SimpleTag
    {
    public JspContext jspcontext;
    public JspTag parentTag;
    public JspFragment jspBody;
    public void setJspContext(JspContext jspContext)
        { this.jspcontext = jspContext;
        }
    public JspContext getJspContext()
        { return jspcontext;
        }
    public JspTag getparent()
        { return parentTag;
        }
    public void setparent(JspTag parentTag)
        { this.parentTag = parentTag;
        }
    public void setJspBody(JspFragment jspBody)
        { this.jspBody = jspBody;
        }
    public JspFragment getJspBody()
        { return jspBody;
        }
```

public void doTag() throws JspException, IOException
{
==
}
}

public class MyHandler extends SimpleTagSupport
{
==
}

---

simpletagapp
- hello.jsp
- WEB-INF
  - hello.tld
  - classes
    - com.dss.Hello.class

hello.jsp

<%@ taglib uri="/WEB-INF/hello.tld" prefix="mytag" %>
<html>
<body bgcolor="

hello.tld

```
<taglib>
<tld-version>
<jsp-version>
<tag>
  <name> hello
  <tag-class> com.dss.Hello
  <body-content> empty <
</tag>
</tag-lib>
```

Hello.java

```
package com.dss;
import javax.servlet.jsp.*;
                .tagext.*;
public class Hello extends SimpleTagSupport
  {
  public void doTag() throws JspException,
                              java.io.IOException
    { try
      {
getJspContext.getOut().println("Hello user");
      }
      catch(exception e)
        {e.printStackTrace();
        }
  }
```

JspContext jspContext

JSTL [JSP Standard Tag Library]

Date: 28th May, 2012 Mon

— In JSP technology scripting elements can be used to provide java code inside the JSP pages. But the main theme of the JSP technology is not to provide java code inside the JSP pages.

To eliminate java code from JSP pages, we have to eliminate scripting elements but to eliminate scripting elements we have to use JSP ACTIONS. In JSP pages, it is possible to use Standard Actions as an alternative to scripting elements but which are in very limited number and having bounded functionality so that Standard Actions are not sufficient to eliminate the complete java code from JSP pages.

— In the above context, to eliminate the complete java code from JSP pages we have to use Custom Actions along with the Standard Actions.

— In case of custom Actions to implement a simple programming constructs like if, for, switch and so on we have to provide a lot of java code internally in the form of Tag Handler classes.

— To overcome the above problem, JSP technology has provided JSTL (JSP Standard Tag Library). In JSTL, JSP technology has provided a set of predefined tags to represent the situations which are not represented by Action tags and where custom tags require lot of java code.

2→

— JSTL is an abstraction provided by sun microsystems but whose implementations are provided by server vendors. With the above convention, Apache Tomcat has provided JSTL implementations in the form of standard.jar file and jstl.jar file at the following location in Tomcat software (in any web application)

C:\Tomcat 7.0\webapps\examples\WEB-INF\lib

↑ any web application name

— JSTL abstraction has provided the following types of tags

1. core tags
2. xml tags
3. formatted tags (I18N tags)
4. SQL tags
5. Functions tags.

2→ If we want to get the JSTL support into our jsp pages first we have to keep standard.jar and jstl.jar file in our web application lib

After getting the required jar files folder in lib folder we have to extract standard.jar file and pickup the required tld files from

META-INF folder can locate at any location of web application directory structure and use the name and location of tld files, as value to uri attribute in taglib directive in our JSP pages.

— To get taglibrary support in jsp pages the above approach is not suggestable, to simplify this process JSTL has defined standard uri's for each and every tag library and suggested to use that uris directly as values to uri attribute in the taglib directives instead of tld file name and location.

— To represent each and every tag library JSTL has provided the following uri's

coreTags → http://java.sun.com/jstl/core
xmltags → https://java.sun.com/jstl/xml
formattags → https://java.sun.com/jstl/fmt
sql tags → https://java.sun.com/jstl/sql
functions tags → https://java.sun.com/jstl/jsp/fn

In the tag of tld file

core tag library :- (core tags)

In JSTL, core tag library has focussed mainly on the general purpose tags, conditional expressions implementations, to represent iterations that is flow control and to perform url based operations, like send Redirect mechanism.

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

① General purpose tags :- <c:set -->, <c:out -->
<c:remove -->, <c:catch -->

② conditional tags :-
<c:if -->, <c:choose -->, <c:when -->, <c:otherwise -->

③ Iterative tags :-
<c:forEach -->, <c:forTokens -->

④ url related tags :-
<c:import -->, <c:url -->, <c:redirect -->

JSTLapp
- core tags.jsp
- WEB-INF
  - jsp
    - jstl.jar
    - standard.jar
  - classes

<C:set-->

Date: 8th may, 2012 Tue

—where scope attribute will take either of the scopes like page, request, session and application.

<C:out--> :- This tag can be used to display a specified value on client browser. [This tag is same as JspWriter]

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

where value attribute will take a value to display on client browser.

ex: <c:out value="AAA"/>

— If we want to display a particular variable value available in either of the scopes then we have to use the following syntax.

<c:out value="${var}"/>

If we want to use the above syntax in Jsp page to access a variable value then we have to eliminate Expression Language syntaxes from the Jsp pages.

For this we have to use the following page directive → why this because container will be in confusion to expose (JSTL) count with EL expr error

<%@page isELIgnored="true"/>

→ If we provide the second type syntax in every JSP page then container will search for the value of the variable in respective scopes in order like, from page, one starting

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

<html>
<body>
<b><font size="7">

<c:set var="a" value="AAA"
scope="application"

pending

③ <c:remove --> tag :- This tag can be used to remove a particular key, value > pair from the either of the scopes page, request, session and application.

```
<c:remove var="__" />
```

where var attribute will take a key to remove from the respective scope.

example
page

```
<c:set var="a" value="AAA" scope="application" />
<c:out value="${a}" />
<c:remove var="a" />
<c:out value="${a}" />
</font></b></center></body></html>
```

<c:catch --> :- This tag can be used to catch the generated exception from the body of <c:catch --> tag. [This tag functionality is almost all equal to the try, catch block]

<c:catch var="_" />

</c:catch>

— where var attribute will take a variable to hold the generated exception.

Expl:

```
<c:catch var="e">
<%= 100/0; %>
</c:catch>
<c:out value="${e}" />
</b></center></body></html>
```

java.lang.ArithmeticException: / by zero ✓

<conditional tags :-

<if --> :-

This tag can be used to implement if programming construct

<c:if test="_"/>

— where test is a boolean attribute, it will take conditional expression.

```
Before if
<br><br>
<c:if test="${10>20}">
  if Body <br><br>
</c:if>
After if
```

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

These three tags can be used to implement switch programming construct.

Syntax

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

Ex:

```
c:set var="a" value="30" />
  <c:choose>
  <c:when test="${a==5}"> FIVE </c:when>
  <c:when test="${a==10}"> TEN </c:when>
  <c:when test="${a==15}"> FIFTEEN </c:when>
<c:when test="${a==20}"> TWENTY </c:when>
  <c:otherwise>
    Number not 5/10/15/20
  </c:otherwise>
```

</c:choose>
<!

Iterative tags:-

<c:forEach --> tag :- This tag can be used to iterate a loop with specified number of times and to retrieve the elements from the specified collection object.

Syntax:

<c:forEach var="_" begin="_" end="_" step="_">

</c:forEach>

- where var attribute will take the variable to represent loop index value.
- begin attribute will take loop start index and end attribute will take loopEnd index value
- step attribute will take increament length.

```
<c:forEach var="a" begin="b" end="p" step="2">
<c:out value="${a}"/><br><br>
</c:forEach>
</font></b></center></body></html>
```

0 2 4 6 8

syntax:②:

```
<c:forEach var="_" items="_">
    ==========
</c:forEach>
```

— where var attribute will take a variable to hold an element retrieved from the specified collection at each and every iteration.

— where items attribute will take the reference of collection available in page, request, session and application.

expl:

```
<%@ page import="java.util.*" %>
<jsp:scriptlet>
String[] str={"aaa","bbb","ccc","ddd"};
request.setAttribute("str",str);
</jsp:scriptlet>
<c:forEach var="a" items="${str}">
<c:out value="${a}"/><br>
</c:forEach>
```

// send this attribute to the request scope

 </body></html>

<c:forTokens-->:-

This tag can be used to perform string tokenization

syntax:

<c:forTokens var="" items="" delims="">
==
</c:forTokens>

where var attribute will take a variable to hold token at each and every iteration.
- where items attribute will take a string to tokenize.
- where delims attribute will take a delimiter to perform tokenization.

<c:forTokens var="token" items="Durga Software solutions"
delims=" ">
↳ space will be there in b/w quotes

<c:out value="${token}"/>

</c:forTokens>

Url related tags

- This tag can be used to include target resource content into the present jsp page.

Syntax: `<c:import url="--"/>`

where url attribute will take the name and location of the target resource.

Eg: coretag.jsp

```
purga<br>
<c:import url="aaa.jsp"/><br>
  Solutions
</font></b></center><
```

aaa.jsp

```
Software
```

<c:url --> :- This tag can be used to display a particular url at web browser.

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

where value attribute will take a particular url.

Eg:

<c:url value="http://localhost:1010/registration app/registrationform.jsp"/>
 </body></html>

— http://localhost:1010/registrationapp/registrationform.jsp.

<c:redirect -->

This tag can be used to implement server redirect mechanism that is it will redirect the request from one web application to another web application

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

where url attribute will take the target url

<c:redirect url="http://localhost:1010/registrationapp/registrationform.jsp"/>
 </body></html>

}

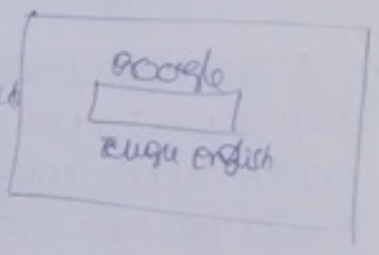
Formatted Tags (I18N):-

Date: 30th may, 2012 Wed

Designing Java application with respect to a particular locale is called as Internationalization. [Google company]

- Java technology is able to provide good internationalization support due to the availability of unicode character representation.
- To provide internationalization support in our Java applications first we have to represent a group of locale users in java applications.

2. To represent a group of locale users, Java technology has provided a predefined class in the form of java.util.Locale



- To create Locale class object we have to use the following constructors
  1. Locale(String lang)
  2. Locale(String lang, String country)
  3. Locale(String lang, String country, String system variant)

① → group of locale people based on lang.

② → group of locale people based on lang and country

③ → represents group of locale people based on lang, country and system-variant

ex: `Locale l = new Locale("en", "US", "WIN");`

Here 'l' represents the group of english people residing at US and uses windows operating system.

— In general from one locale to another locale numbers representations will be vary. To carry out a particular number w.r.to a locale, java api has provided a predefined class

java.text.NumberFormat

— In general from one locale to another locale Dates representations will be vary. To represent a particular date w.r.to a Locale then java technology has provided a predefined class. That is

java.text.DateFormat

— From one locale to another locale, messages representations will be vary. To represent a particular message with respect to a Locale, java technology has provided a predefined class in the form of

java.util.ResourceBundle

— To provide Internationalization support in web applications JSTL has provided a separate tag library called as Formatted Tags

— JSTL has provided the following tags as part of formatted tags to provide Internationalization Support.

① <fmt:SetLocale -->
② <fmt:formatNumber -->
③ <fmt:formatDate -->
④ <fmt:SetBundle --> / <fmt:bundle>
⑤ <fmt:message -->

<fmt:setLocale />

This tag can be used to specify a particular locale in the present jsp page.

```
<fmt:setLocale value="_" />
```

Here value attribute will take language and country locale parameters.

ex:

```
<fmt:setLocale value="en_US" />
```

This is equivalent to

```
Local l = new Locale("en","US");
```

— In jsp applications, if we have not used <fmt:setLocale> tag then container will take browser provided locale (that means the settings defaultly used by browser)

② <fmt:formatNumber /> : This tag can be used to represent a particular number in the specified locale.

Syntax:

```
<fmt:formatNumber value="" />
```

where value attribute will take a particular number.

example Numbers.jsp

```
<%@ taglib uri="http://java.sun.com/jstl/fmt"
    prefix="fmt" %>
<html>
<body>
<center><b><font size="4"><br><br>
<fmt:setLocale value="pt_IT" />
```

<fmt:formatDate>:-

- This tag can be used to represent a date w.r.t a particular locale.

Syntax: <fmt:formatDate value="" type=""/>

- where value attribute should take Date object reference and type attribute will take either Date or time to display current date and time

```
<%@taglib url=

<%@page isELIgnored="true" %>
  <html>
   <body>
   <center><b><font size="7"><br><br>
   <fmt:setLocale value="pt-IT"/>
   <jsp:useBean id="date" class="java.util.Date">
   <fmt:formatDate value="${date}" type="time"/>
   </jsp:useBean>
   </font>
```

<fmt:setBundle --> / <fmt:bundle -->

These tags can be used to represent a particular properties file.

Syntax: <fmt:setBundle baseName="L"/>

If we want to use this tag in jsp page, we should require the properties files with required key value pair under classes folder. To save a properties file we should use the following convention.

baseName_<lang:Name>_<Country_Name>.properties

ex: abc_en_US.properties ✓
abc_pt_IT.properties ✓

<fmt:message -->

This tag can be used to get a particular message from the properties file represented by <fmt:setBundle --> tag on the basis of the provided key.

syntax: <fmt:message key="L"/>

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

abc_en_US.properties:

welcome = Welcome TO en US user.

abc_pt_IT.properties

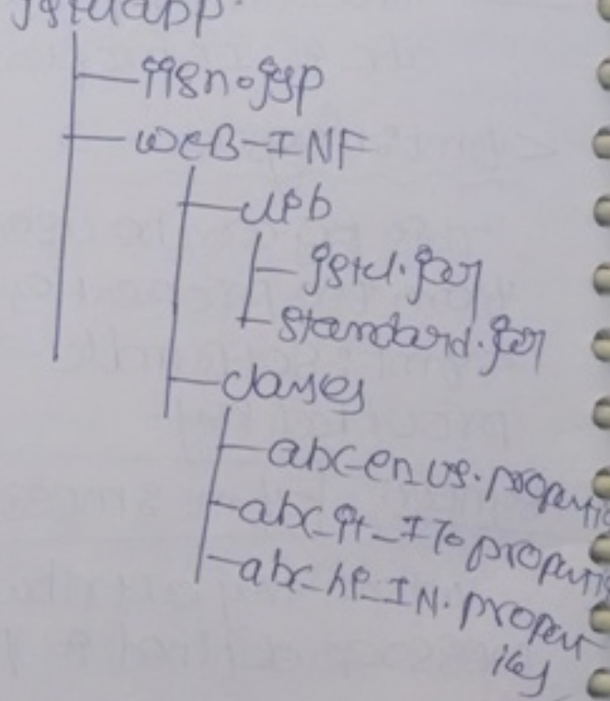
welcome = welcome To Pt IT user.

abc_hi_IN.properties

welcome = welcome to hi IN user.

I18n.jsp

```
<%@ taglib url="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<html>
<body>
<center><b><font size="7"><br><br>
<fmt:setLocale value="it-IT"/>
<fmt:setBundle baseName="abc"/>
<fmt:message key="welcome"/>
</font></b></center></body></html>
```



SQL Tags :- The main purpose of the SQL tag library is to interact with the database in order to perform the respective database operations.
- To perform the database operations from JSP pages directly, JSTL has provided the following tags as part of SQL Tag library.

① <sql:setDataSource -->
② <sql:update -->
③ <sql:query -->
④ <sql:param -->
⑤ <sql:dateparam -->
⑥ <sql:transaction -->

<sql:setDataSource -->:- This tag can be used to establish the connection with the database by taking all the jdbc parameters.

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

where driver and url attributes will take driver class name and driver url.
- where user and password attributes will take database user name and password.

ex: <sql:setDataSource driver="oracle.jdbc.driver.OracleDriver"
url="jdbc:oracle:thin:@localhost:1521:xe"
user="system"
password="durga" />

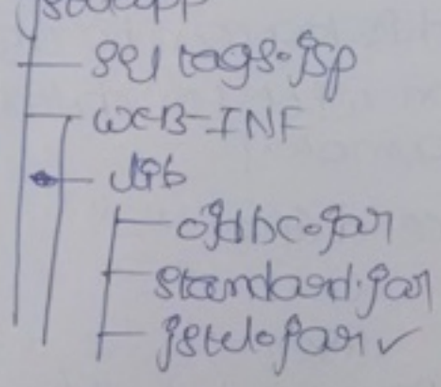
Date: 31st may, 2012 Thurs

<sql:update ---> :- this tag can be used to execute all the updation group sql queries like create, insert, update, delete, ---

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

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

- where var attribute will take a variable name to hold row count value generated as a result of the specified sql query execution. particular
- where sql attribute will take the specified sql query to execute.

ex:


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

<%@ page isELIgnored="true" %>

<sql:setDataSource driver="oracle.jdbc.driver.OracleDriver" url="jdbc:oracle:thin:@localhost:1521:xe" user="system" password="durga" />

<sql:update var="rowCount" sql="create table emp(eno number, ename varchar2(5), esal number)" />

```
<html>
<body>
<center><b>
<hr><br>
RowCount--- <c:out value="${rowCount}"/>
</font></b></center></body></html>
```

Ex②: sqltags.jsp

```
<%@ taglib url="http://java.sun.com/jsp/jstl/sql"
                    prefix="sql" %>
<%@ page isELIgnored="true" %>
<sql:setDataSource driver="oracle.jdbc.driver.oracle
Driver" url="jdbc:oracle:thin:@localhost:1521:xe"
user="system" password="durga"/>

<sql:update var="rowCount">
insert into emp values(222,'bbb',600)
</sql:update>
<html>
<body>
<center><b><hr><br>
RowCount---<c:out value="${rowCount}"/>
</font></b></center></body></html>
```

2→ Incase of <sql:update--> tag, it is possible to provide the sql query with the positional parameters as per prepared statement.

— If we provide positional parameters in sql query then we set values to positional parameters we have to use the following tag as a child tags to <sql:update-->

Syntax①: <sql:parameter value="--"/>

Syntax②: <sql:parameter>value</sql:parameter>

2→ If we want to provide a particular date as parameter to Date parameter then we have to use <sql:dateparam> tag

ex③

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql"
                                    prefix="sql" %>
<%@ page isELIgnored="true" %>
<sql:setDataSource driver="oracle.jdbc.driver.
OracleDriver" url="jdbc:oracle:thin:@localhost:1521:
xe" user="system" password="donga" />
<sql:update var="noOfRowsInserted" sql="insert into
    emp values(?,?,?)">
<sql:param value="333" />
<sql:param value="ccc" />
<sql:param>7000</sql:param>
</sql:update>
  <html>
    <body>
    <center><b><hr><br>
```

Rowcount --- <c:out value="${rowCount}"/>
 </body></html>

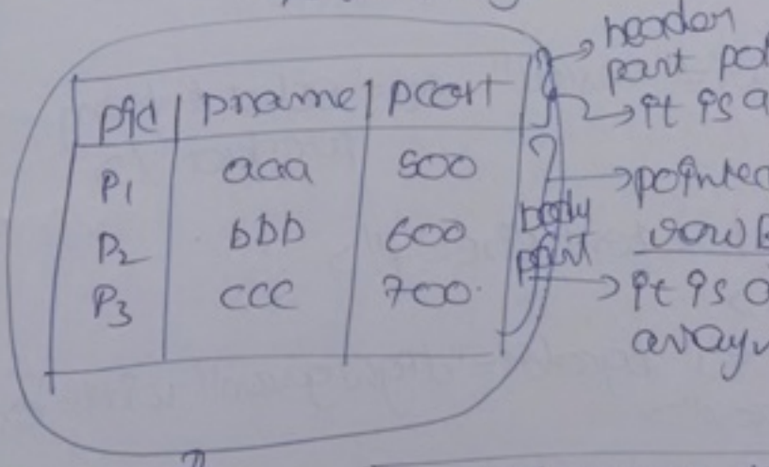
Ans

<sql:update var="rowCount">
update emp set esal=esal+? where esal>?
<sql:param value="500"/>
<sql:param>500</sql:param>
</sql:update>
<html>
<body>

— <sql:query --> :- This sql tag can be used to execute selection group sql query in order to fetch the data from database table.

sql①: <sql:query var="" sql="" />

sql②: <sql:query var="">
— sql query —
</sql:query>



Result (not resultset object)

① <sql:query var="result" sql="select * from product" />

It will execute the above sql query internally all the records will be retrieved from database table and the overall result

will be store in the form of result object.

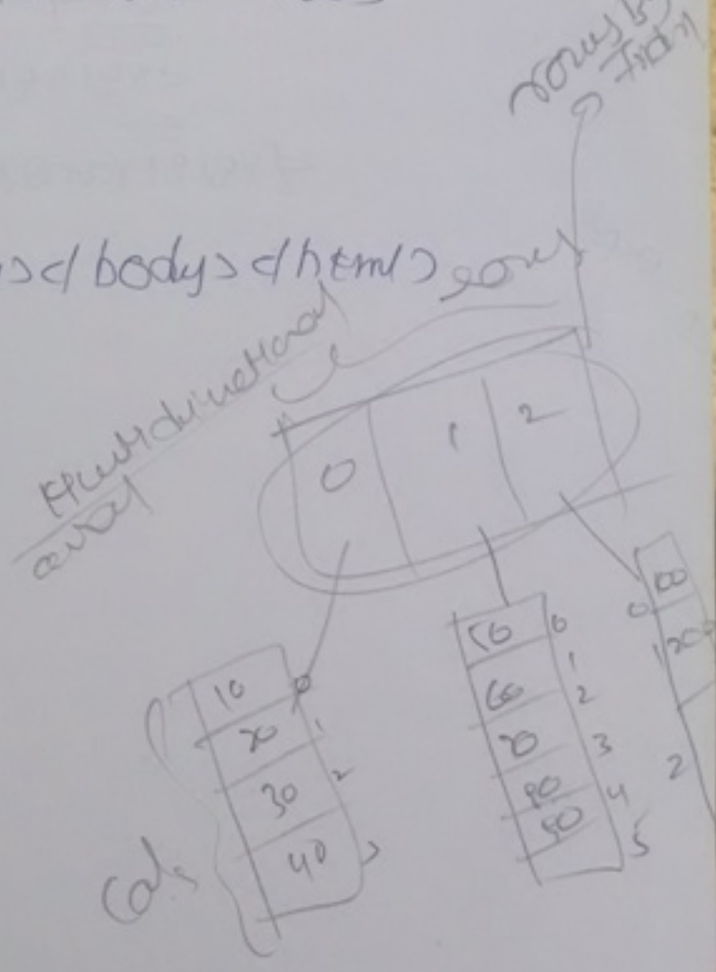
— In result object, all the column names will be represented in the form of a single dimensional array represented by a predefined variable column names.

In the result object, all the records data will be stored in the form of a dimensional array represented by a predefined variable rows by Index.

```
<%@ taglib url="http://java.sun.com/jstl/sql"
        prefix="sql" %>
<%@ taglib url="http://java.sun.com/jstl/core"
        prefix="c" %>
<sql:setDataSource driver="
```

```
<sql:query var="result" sql="select * from
        product" />
<html>
  <body>
  <center><b><font size="7">
    <br><br>
  <table border="1" bgcolor="lightgreen" width="30%"
    height="30%">
    <tr>
  <c:forEach var="column" items="${result.
                columnNames}">
  <td><c:out value="${column}" />
                </td>
```

```
</c:forEach>
</tr>
<c:forEach var="row" items="${result.rowsByIndex}">
<tr>
<c:forEach var="col" items="${row}">
<td><c:out value="${col}" /></td>
</c:forEach>
</tr>
</c:forEach>
</table>
</font></b></center></body></html>
```



<sql:transaction --> tag:

This sql tag will represent a single transaction that is an unit of work. This tag will include number of <sql:update> tags and <sql:query> tags as per the requirement ✓

Syntax

```
<sql:transaction>
    <sql:update />
    ==========
    <sql:query />
    ==========
</sql:transaction>
```

Example

Expression Language

- In general the main intention of jsp technology is to reduce java code as much as possible from web applications. To achieve the above requirement jsp technology has provided number of most alternatives like implicit objects, standard actions, custom actions, jstl tag library and so on.
- Even though if we have number of alternatives in jsp technology still it is required to provide some java code in jsp pages inorder to achieve client requirements. In the above context to eliminate java code completely from jsp pages we have to use expression language syntax. make

ex: To retrieve a particular parameter value from request object jsp technology has not provided any alternative, we must provide scripting elements.

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

- For the above requirement, standard actions are not provided in any environment, jstl tag library has not provided any alternative but expression Language has provided an alternative

uname: abc

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

- In expression language, we often provide any expression in the form of the following syntax

```
${expression}
```

→ In jsp technology, expression language is available in the form of the following elements

① operators
② implicit objects
③ EL-FUNCTIONS ✓

operators :-

① General purpose operators :-
(i) . ⟶ to access a variable or property.
(ii) {} ⟶ to represent expression.
(iii) [] ⟶ to represent an array.
(iv) () ⟶ to represent parameters.

② Arithematic operators :-
+, -, *, /, %, ++, --

③ comparison operators :-
== (or) eq
< (or) lt
> (or) gt
<= (or) le
>= (or) ge
=

④ Logical operators
! (or) not
& (or) and
| (or) or

⑤ Implicit objects :-
Like jsp implicit objects, Expression language has provided its own set of implicit objects.

① page scope
② request scope
③ session scope.
④ application scope.
⑤ param.
⑥ paramValues
⑦ initparam.
⑧ header.
⑨ headerValues
⑩ cookie.
⑪ pageContext

— where page scope, request scope, session scope and application scope implicit objects can be used to access the attributes which are available in the respective scope.

Ex:

```
<% request.setAttribute("uname", "Durga");
%>
<html><body><center><h1>
UserName - ${requestScope.uname}
</h1></
```

@p

2→ where param implicit object can be used to access a particular request parameter from request object. where param values implicit object can be used to
which are associated with a single request parameter.

Ex: form.html

```
<html><body bgcolor="Sequpink"><br><br><br>
<font size="7">
<form method="get" action="./e1.jsp">
UserName% <input type="text" name="uname"/>
password% <input type="text" name="upwd"/>
<br><br>
qualifications: <input type="checkbox"
name="uqual" value="BSc"/> BSc
<input type="checkbox" name="uqual"
value="MCA"
```

a.jsp

```
<html><body><center><font size="6">
User Name --- ${param.uname}
<br><br>
password --- ${param.upwd}
<br><br>
Qualification -- ${paramValues.uqual[0]}
${paramValues.uqual[1]}
${paramValues.uqual[2]}
</font></center></body></html>
```

Syntax: ${initparam.driver}

⟹ where header implicit object can be used to access a single request header value. where header implicit object can be used to access multiple number of values which are associated with single request header

⟹ cookie implicit object can be used to get sessionId cookie name and value.

ex:

```
<h1>
${cookie.JSESSIONID.name}
<br><br>
${cookie.JSESSIONID.value}
</h1>
```

⟹ pageContext implicit object is almost all same as jsp pageContext implicit object, which can be used to make available all the other jsp implicit objects

EL-functions :- Incase of the custom tags, to define a particular custom tag we have to provide the required java code in the form of tag handler class, to prepare taghandler class we have to use a predefined library provided by jsp technology.

— similarly, if we want to perform an action in jsp pages without using tag library we have to use expression Language functions.

— To define expression Language functions in jsp technology we have to use the following steps

1. Define a static function in a java class under classes folder.

```
class MyClass
{ public static String myFun()
  {
  }
}
```

② — configure this function class and function name in the respective tld file. To achieve this we have to use the following tags.

```
<taglib>
  ...
  <function>
    <name>function_name</name>
    <function-class> fully qualified
                     name of the
                     function class
                     </function-class>
    <function-signature>
    function signature </function-signature>
  </function>
</taglib>
```

③ Access the function from jsp page.

${ fn:myFunction([param-list]) }

elfunctionapp
- el.jsp
- WEB-INF
  - el.tld
  - classes
    - com.dss.Hello.class

Hello.java : package com.dss

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

el.tld :

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</tlib-version>
  <function>
   <function-name>sayHello</name>
   <function-class>com.dss.Hello</functionclass>
   <function-signature>java.lang.String
   sayHello(java.lang.String)</function-signature>
  </function>
  </taglib>
```