Advanced Java

DURGA SOFTWARE SOLUTIONS

JSP Syllabus

j
s
p

Detailed Syllabus

Unit-1: The Java Server Pages (JSP) Technology Model

Objectives

- Identify, describe, or write the JSP code for the following elements: (a) template text, (b) scripting elements (comments, directives, declarations, scriptlets, and expressions), (c) standard and custom actions, and (d) expression language elements.
- Write JSP code that uses the directives: (a) 'page' (with attributes 'import', 'session', 'contentType', and 'isELIgnored'), (b) 'include', and (c) 'taglib'.
- Write a JSP Document (XML-based document) that uses the correct syntax.
- Describe the purpose and event sequence of the JSP page life cycle: (1) JSP page translation, (2) JSP page compilation, (3) load class, (4) create instance, (5) call the jspInit method, (6) call the _jspService method, and (7) call the jspDestroy method.
- Given a design goal, write JSP code using the appropriate implicit objects: (a) request, (b) response, (c) out, (d) session, (e) config, (f) application, (g) page, (h) pageContext, and (i) exception.
- Configure the deployment descriptor to declare one or more tag libraries, deactivate the evaluation language, and deactivate the scripting language. 6.7 Given a specific design goal for including a JSP segment in another page, write the JSP code that uses the most appropriate inclusion mechanism (the include directive or the jsp:include standard action).

Unit- 2: Building JSP Pages Using the Expression Language (EL)

Objectives

- Given a scenario, write EL code that accesses the following implicit variables including pageScope, requestScope, sessionScope, and applicationScope, param and paramValues, header and headerValues, cookie, initParam and pageContext.
- Given a scenario, write EL code that uses the following operators: property access (the . operator), collection access (the [] operator).

Unit-3: Building JSP Pages Using Standard Actions

Objectives

- Given a design goal, create a code snippet using the following standard actions: jsp:useBean (with attributes: 'id', 'scope', 'type', and 'class'), jsp:getProperty, jsp:setProperty (with all attribute combinations), and jsp:attribute.
- Given a design goal, create a code snippet using the following standard actions: jsp:include, jsp:forward, and jsp:param

Unit-4 : Building JSP Pages Using Tag Libraries (JSTL)

Objectives

- For a custom tag library or a library of Tag Files, create the 'taglib' directive for a JSP page.
- Given a design goal, create the custom tag structure in a JSP page to support that goal.
- Given a design goal, use an appropriate JSP Standard Tag Library (JSTL v1.1) tag from the "core" tag library.

Unit-5 : Building JSP Pages Using Custom Tag Library

- Describe the semantics of the "Classic" custom tag event model when each event method (doStartTag, doAfterBody, and doEndTag) is executed, and explain what the return value for each event method means; and write a tag handler class.
- Using the PageContext API, write tag handler code to access the JSP implicit variables and access web application attributes.
- Given a scenario, write tag handler code to access the parent tag and an arbitrary tag ancestor.
- Describe the semantics of the "Simple" custom tag event model when the event method (doTag) is executed; write a tag handler class; and explain the constraints on the JSP content within the tag.
- Describe the semantics of the Tag File model; describe the web application structure for tag files; write a tag file; and explain the constraints on the JSP content in the body of the tag.

Java Server Pages (JSP)

Need Of JSPs:

Servlet technology introduced in 1997.
It is best suitable for processing logic but not for presentation logic.

Usage of Servlet for presentation Logic is not recommended. Even for small output also, we have to write huge code with Strong java knowledge.

To overcome this problem SUN people introduced JSP Technology in 1999. JSP is best suitable for presentation logic (View Component).

Servlet vs JSP:

Servlets meant for processing logic.
Eg: verify user
check account balance
collect data from database
i.e where ever some processing is required then we should go for Servlets.

JSP meant for Presentation logic. i.e. to display something to end user we should go for JSPs.
Eg: display login page
display inbox page
display error page

Note:

1. Inside Servlets we should not write presentation logic. i.e. we are not allowed to use out.println() statements inside servlet. If any person writing then cut his hand.

2. Inside JSP we should not write business code/processing code. We should write only presentation logic. i.e. we are not allowed to write any java code inside JSPs. If any person writing then cut his head.

Versions:

Servlets and JSPs are part of Java Enterprise edition(JEE | J2EE)

Current version of JEE is 7
Servlets ➔ 3.1V
JSPs ➔ 2.3V
JSTL ➔ 1.2V
EL ➔ 3.0V

Unit 1: JSP Technology Model

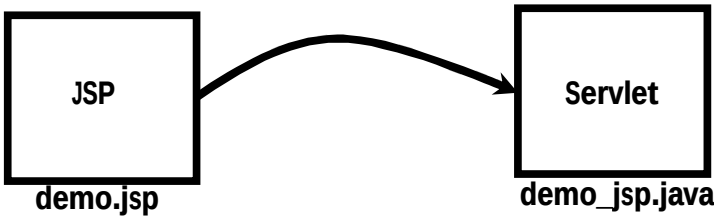
JSP API and Life Cycle:

Objective:

Describe the purpose and event sequence of JSP Page Life Cycle

1. Translation phase
2. JSP Page Compilation
3. Load the class
4. Create an instance
5. Call jspInit() method
6. call _jspService() method
7. call jspDestroy() method

The process of Translating JSP Page (.jsp file) into corresponding Servlet (.java file) is called Translation Phase.



This can be done by JSP Engine. In Tomcat this component is called JASPER.

demo.jsp:

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

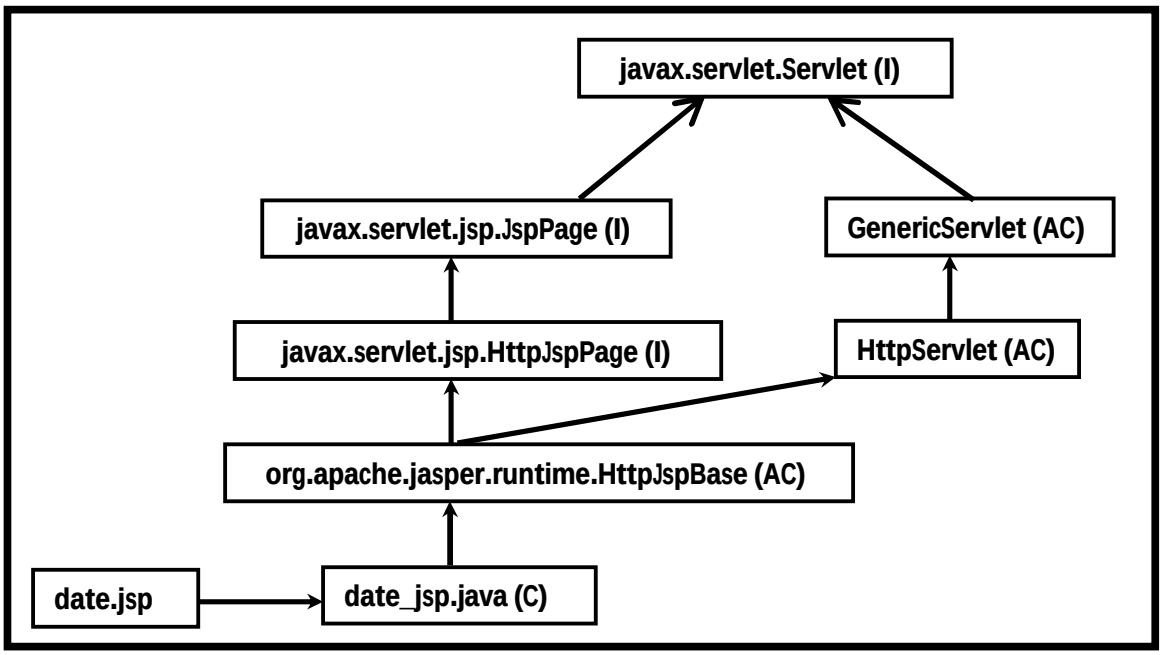
The generated .java file is available in the following location in Tomcat

D:\Tomcat 7.0\work\Catalina\localhost\jspdapp1\org\apache\jsp\demo_jsp.java

JSP API:

Every servlet which is generated for the JSP must implements javax.servlet.jsp.JspPage(I) OR javax.servlet.jsp.HttpJspPage(I) either directly or indirectly.

Tomcat people provided a Base class *org.apache.jasper.runtime.HttpJspBase* for implementing the above 2 interfaces. Hence any servlet which is generated by Tomcat is extending this HttpJspBase class.



JSP API defines the following 2 interfaces.

1. JspPage
2. HttpJspPage

I) JspPage(I):

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

1) jspInit()
2) jspDestroy()

1) jspInit():

```
public void jspInit()
```

- This method will be executed only once at the time of first request to perform initialization activities.
- Web container always calls init(ServletConfig) of HttpJspBase class which internally calls jspInit() method.

```
1)  public abstract class HttpJspBase ..
2)  {
3)    public final void init(ServletConfig config)..
4)    {
5)       jspInit();
6)    }
7)  }
```

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

test.jsp:

```
1)  <%!
2)   public void jspInit()
3)   {
4)      System.out.println("jsp initialization activities");
5)   }
6)  %>
7)  <h1>The Server Time is:<%= new java.util.Date() %></h1>
```

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

2) jspDestroy():

```
public void jspDestroy()
```

- This method will be executed only once to perform cleanup activities just before taking jsp from out of service.

- We can override jspDestroy() method in our jsp to define our own cleanup activities.
- Web container always calls destroy() method available in HttpJspBase class which internally calls our jspDestroy() method.

```
1)  public abstract class HttpJspBase ..
2)  {
3)    public final void destroy()
4)    {
5)       jspDestroy();
6)    }
7)  }
```

test.jsp:

```
1)  <%!
2)   public void jspDestroy()
3)   {
4)      System.out.println("jsp cleanup activities");
5)   }
6)  %>
7)  <h1>The Server Time is:<%= new java.util.Date() %></h1>
```

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

II) HttpJspPage(I):

- It is the child interface of JspPage.
- It defines only one method _jspService() method.

```
public void _jspService(HSR req,HSR resp)throws SE,IOE
```

- This method will be executed seperately for every request.
- Web container always calles service() method present in HttpJspBase class which internally calls our service()method.

```
1)  public abstract class HttpJspBase extends..
2)  {
3)     public final void service(HSR request, HSR response) throws SE, IE
4)     {
5)        _jspService(request, response);
6)     }
7)  }
```

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

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

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

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

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

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

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

Note:

JspPage➔jspInit(),jspDestroy()
HttpJspPage➔ _jspService()

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

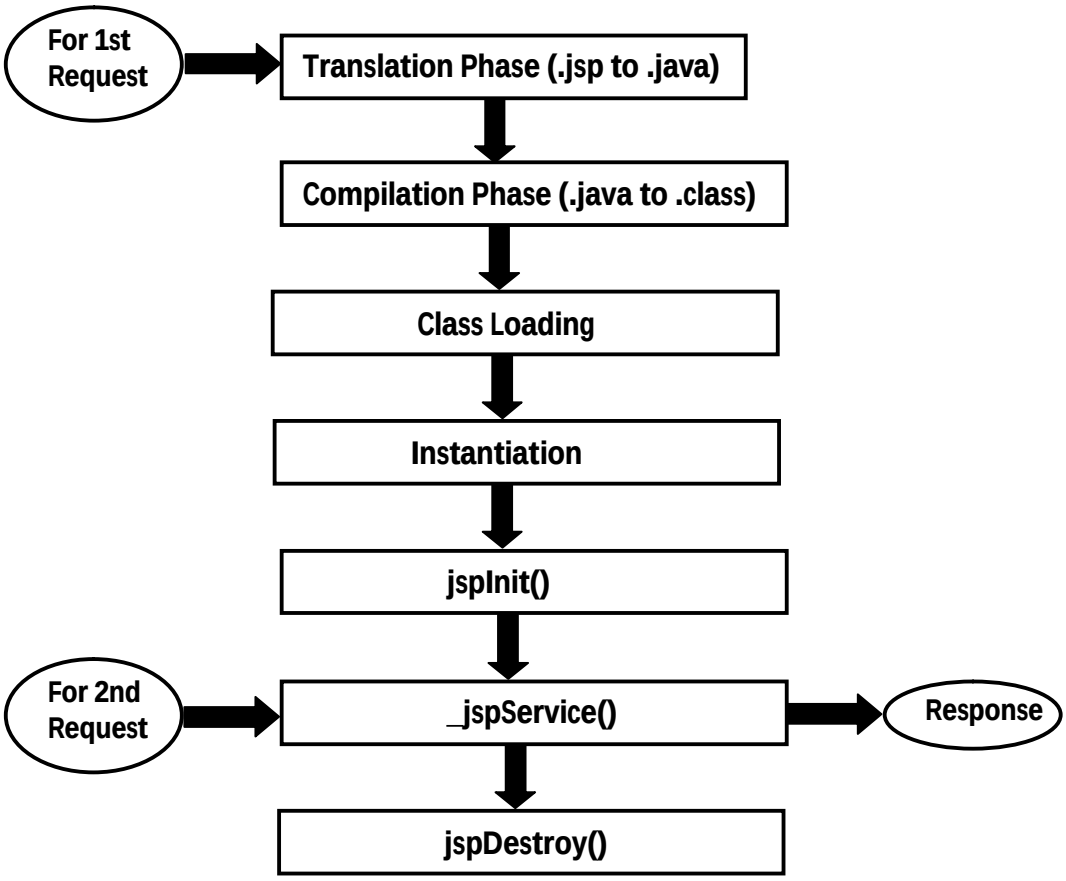
1.init() ✕
2.jspInit() ✔
3.destroy() ✕

4.jspDestroy() ✔
5.service() ✕
6._jspService() ✕

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

Life Cycle of JSP:

- Translation Phase (.jsp➔.java)
- Compilation Phase(.java ➔.class)
- Servlet Class Loading
- Servlet Instantiation
- jspInit()
- _jspService()
- jspDestroy()



***Note:

JSP will participate in Translation Phase in the following cases

1. At the time of First Request

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

Pre Compilation of JSP:

In the case of JSP, at the time of first request the following activities will be performed.

- Translation
- Compilation
- class Loading
- Instantiation
- jspInit()
- _jspService()

But for second request onwards only _jspService() method will be executed. Hence the processing time of first request is more when compared with remaining requests.

To overcome this problem we should go for pre compilation of JSP.

We can invoke pre compilation as follows..

http://localhost:7777/jspdapp1/demo.jsp?jsp_precompile=true

It is not request to jsp and it is just performing pre compilation. B'z of this the following activities performed

- Translation
- Compilation
- class Loading
- Instantiation
- jspInit()

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

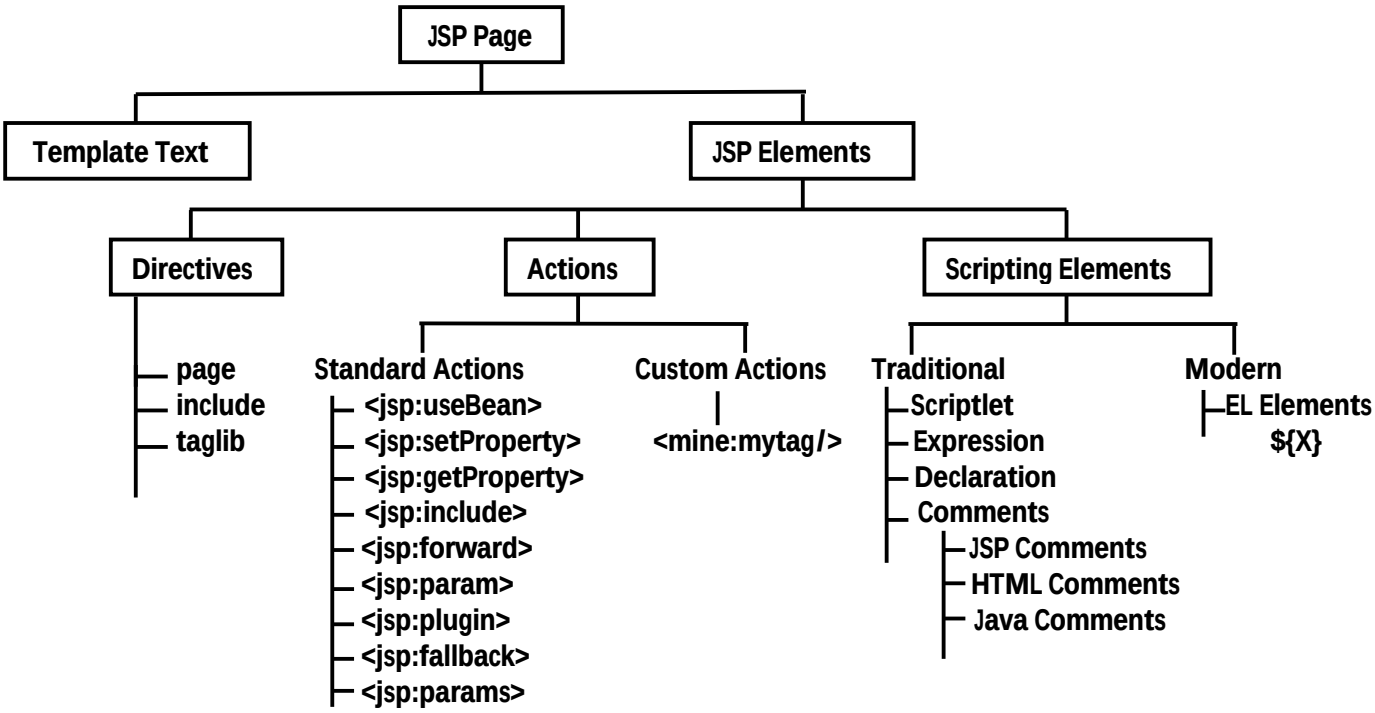
http://localhost:7777/jspdapp1/demo.jsp

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

JSP Scripting Elements

Objective: Identify, Describe, write JSP Code for following elements

1) Template Text
2) Scripting Elements
3) Standard and Custom actions
4) Expression Language Elements



Template Text:

It contains plain text data and html, xml tags.

For the template text no processing is required and it will become argument to out.write() method in _jspService() method.

test.jsp:

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

For the above jsp the generated servlet class is:

```
1.  public final class demo_jsp extends...
2.  {
3.    ...
```

```
4.    public void _jspService(..)...
5.    {
6.       out.write("<h1>The Server Time is:");
7.       out.print(new java.util.Date());
8.       out.write("</h1>");
9.    }
10. }
```

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

write() method can take only character data as argument

print() method can take any type of data as argument

Template text is always character data and hence it will become argument to write() method.

But expression value can be any type and hence we should not use write() method, compulsory we should go for print() method which can take any type of argument.

Directives:

These can provide general information about JSP page to the JSP Engine. i.e Directives are translation time instructions to the JSP Engine.

Syntax: <%@ directiveName attributeName=attributeValue %>

There are 3 types of directives are possible

1. page directive:

   <%@ page attributeName=attributeValue %>

   It can be used to define page specific attributes.

2. include directive:

   <%@ include file="header.html" %>

   To include the content of header.html at translation time

3. taglib directive:

   <%@ taglib prefix="mine" uri="www.durgasoft.com" %>

To make custom tags available to our JSP

Actions:

Actions are commands to the JSP Engine to perform certain task at runtime(execution time). There are 9 standard actions are available.

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>

Scripting Elements:

There are 2 Types of Scripting Elements

1. Traditional Scripting Elements
2. Modern Scripting Elements

1. Traditional Scripting Elements

- Expression
- scriptlet
- Declarative
- Comments

1. Expression:

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

Syntax: <%= x %>

The equivalent generated servlet code is:

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

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

Eg: test.jsp

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

http://localhost:7777/jspdapp1/test.jsp?user=durga&pwd=sunny

Conclusions:

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

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

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

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

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

Eg: <% =10 %> invalid

the equivalent generated servlet code is:

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

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

<%= String s ="durga" %> invalid

Q.Which of the following are valid java expressions?

1) <%= 27 %> ✔
2) <%= "27" %> ✔
3) <%= Math.random() %> ✔
4) <%= 10*20 %> ✔
5) <%= 10>20 %> ✔
6) <%= new Student() %> ✔
7) <%= String s ="durga" %> ✕

2. Scriptlet:

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

Syntax: <%
Any Java Code
%>

Java Code inside scriptlet will be placed directly inside _jspService() method of generated servlet.
Every java statement present inside scriptlet should compulsary ends with ;

Q. Write a JSP Print hit count of the JSP?

test.jsp:

```
1.  <h1>
2.  <%
3.     int count=0;
4.     count++;
5.     out.println(count);
6.   %>
7.  </h1>
```

It always prints same value 1. b'z count is local variable of _jspService() method

test.jsp:

```
1.  <h1>
2.  <%!
3.     int count=0;
4.  %>
5.  <%
6.     count++;
7.     out.println(count);
8.   %>
9.  </h1>
```

It prints hit count of the jsp and count is instance variable.

Q. What is the difference between the following?

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

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

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

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

3. Declaration Tag:

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

Syntax:
```
<%!
        Any java declarations
%>
```
These declarations will be placed directly in the generated servlet but outside of _jspService() method.

```
1)  <%!
2)    int x =10;
3)    static int y = 20;
4)    int[] a ={10,20,30};
5)    public void m1()
6)    {
7)    }
```

```
8)  %>
```

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

demo.jsp:

```
1)  <%!
2)    public void m1()
3)   {
4)   out.println("Hello");
5)   }
6)  %>
```

CE: out cannot be resolved.

***Note:**

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

But Scriptlet and expression tags code will be placed inside _jspService() method. Hence inside scriptlet and expression tags we can use JSP Implicit objects.

Eg:

```
1)  <%
2)        out.println("Hello");
3)  %>
4)
5)       <%= session.getId() %>
```

4. Comments:

In the JSP, 3 types of comments are allowed.

1. JSP Comments
2. HTML Comments
3. Java Comments

1. JSP Comments:

<%-- This is JSP Comment --%>

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

2. HTML Comments:

<!-- This is HTML Comment -->

Also known as Template text comments.
These are visible to the end user as the part of generated response source code.
Hence these are not recommended to use in the JSP.

3. Java Comments:

```
// single line java comment

/*
 multiline java comment
*/


/**
  Java Doc comment
*/
```

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

*****Note:

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

Summery of JSP Comments

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

Eg1:

```
1)  <%
2)          out.print(<%= new java.util.Date() %>);
3)  %>
```

Eg 2:

```
1)  <%!
2)          <%-- This is very important method.. --%>
3)          public void m1()
4)          {
5)          }
6)  %>
```

Eg3:

```
1)  <%
2)          <%-- This java code meant for user validataion --%>
3)          out.println("validation");
4)  %>
```

Eg 4:

```
1)  <%
2)          <%
3)              out.println("Hello");
4)          %>
5)  %>
```

Comparison Table of JSP Scripting Elements

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

Directives (in Detail)

Objective:

Write JSP Code that uses the following directives
1. page 2. include 3. taglib

Directives can provide general information about JSP page to the JSP Engine. i.e Directives are translation time instructions to the JSP Engine.

Syntax: <%@ directiveName attributeName=attributeValue %>

page|include|taglib

There are 3 types of directives are possible
1) page directive
2) include directive
3) taglib directive

I) page directive:

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

- Eg1: <%@ page language="java" %>
  This page directive specifies to the JSP Engine that the scripting language used in this jsp is java.

- Eg2: <%@ page session="true" %>
  This page directive specifies that the current jsp participating in session management.

- We can use page directive any number of times, anywhere in the jsp.
- The following is the list of all possible 13 attributes of page directive.

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

1) import:

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

demo.jsp without import attribute:

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

demo.jsp with import attribute:

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

To import multiple packages the following are various possibilities...

1)
```
<%@ page import="java.util.*" %>
<%@ page import="java.io.*" %>
```

2)
```
<%@ page import="java.util.*" import="java.io.*" %>
<%@ page import="java.util.*,java.io.*" %>
```

3)
```
<%@ page import="java.util.*" import="java.io.*" %>
```

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

Q. Which of the following page directives are valid?

1) `<%@ page session="true" session="false" %>` ✗
2) `<%@ page session="true" session="true" %>` ✔
3) `<%@ page session="true" %>` ✔
4) `<%@ page import="java.util.*" import="java.io.*" %>` ✔
5) `<%@ page import=java.util.* %>` ✗ B'Z single or double quotes are required.

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

- java.lang.*;
- javax.servlet.*;
- javax.servlet.http.*;
- javax.servlet.jsp.*;

2) session:

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

<%@ page session="false" %>

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

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

If we declare explicitly with false value, then the above code won't be generated.
The allowed values for the session attribute are:

true ✔ TRUE ✔ True ✔ FALse ✔ durga ✕

i.e. case insensitive String of true or false is allowed.
If we are taking any other value then we will get Translation Time error.

Eg: <%@ page session="sunny" %>
Error: invalid value for session

3) contentType:

We have to use this attribute to specify content type of response(i.e MIME Type of Response)

<%@ page contentType="application/pdf" %>

The default value content type is : text/html

4) isELIgnored:

Expression Language(EL) has introduced in JSP 1.2V
The main objective of EL is to remove java code from the JSP.

1. isELIgnored="true"

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

2. isELIgnored="false":

   EL Syntax will be processed and print its value.
   Eg: http://localhost:7777/jspdapp1/demo.jsp?user=Mallika

demo.jsp:

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

o/p: User Name: Mallika

demo.jsp:

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

o/p: User Name: ${param.user}

Note:
The default value of isELIgnored is true in JSP 1.2V.
But from JSP 2.0V onwards the default value is false.

5) isThreadSafe:

By Default a single JSP page can be accessed by multiple threads simultaneously. Hence JSP is not thread safe by default.
To provide Thread Safety to the JSP we should go for isThreadSafe attribute.

1) isThreadSafe="true":

It means JSP page is already thread safe & it is not required generated servlet to implement SingleThreadModel.
In this case JSP Page can process any number of client requests simultaneously.

2) isThreadSafe="false":

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

Note: The default value of isThreadSafe attribute is true.

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

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

Summary of Page Attributes

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

II) include directive:

- If several JSPs contain same code then it is recommended to separate that common code in a separate file. Wherever that common code is required just we have to include that file.
- This mechanism is called inclusion mechanism and we can use very commonly to include header and footer information which is common in every JSP.
- The main advantages of Include mechanism are:

  1) It promotes code reusability
  2) It improves maintainability of the code
  3) Enhancement will become very easy

- We can implement include mechanism either by include directive or by include action.

Include Directive:

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

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

Eg:

first.jsp

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

second.jsp:

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

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

Include Action:

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

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

first.jsp

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

second.jsp:

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

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

Differences b/w Include Directive and Include Action

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

Logo Inclusion (Static Include by Include Directive)	
About Us Inclusion (Static Include by Include Directive)	Sports Info Inclusion (Dynamic Include by Include Action)
Politics Information (Dynamic Include by Include Action)	Movies Information (Dynamic Include by Include Action)
Copy Right Inclusion (Static Include by Include Directive)	

Note:

To include the content of header.jsp at translation time, the required import is:

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

To include the response of header.jsp at request processing time (run time), the required import is:

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

Q. Which of the following inclusions are valid?

1.<%@ include page="header.jsp" %> ✗
2.<%@ include file="header.jsp" %> ✔
3.<jsp:include file="header.jsp" /> ✗

4.<jsp:include page="header.jsp" /> ✔
5.<jsp:include page="header.jsp" flush="true"/> ✔
6.<%@ include file="header.jsp" flush="true"%> ✕

Note:

page attribute is applicable for include action where as file attribute is applicable for include directive.

flush attribute is applicable only for include action but not for include directive.

III) taglib directive:

- We can use taglib directive to make custom tags available to our jsp.

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

- taglib directive contains 2 attributes prefix and uri.
- uri represents the location of TLD File which in turn represents TagHandler class.

JSP Implicit Objects

Objective:

For the given design goal write the JSP Code using appropriate implicit objects.?

When compared with Servlet Programming, Developing JSPs is very easy because the required mandatory stuff automatically available in every JSP. Implicit objects also one such area ,which are by default available for every JSP.

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

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

1) Request and Response Implicit Objects:

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

request.jsp:

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

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

2) Application Implicit Object:

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

application.jsp:

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

web.xml:

```
1) <web-app>
2)   <display-name>JSP Implicit Objects Application</display-name>
3)   <context-param>
4)     <param-name>uname</param-name>
5)     <param-value>scott</param-value>
6)   </context-param>
7) </web-app>
```

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

3) Session Implicit Object:

In every JSP session implicit object is by default available and it is of type HttpSession.
Whatever methods we can apply on HttpSession object, all those methods we can apply on session implicit object.

session.jsp:

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

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

test.jsp:

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

CE: session cannot be resolved.

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

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

4) Config Implicit Object:

config object is of type ServletConfig.

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

The following is the list of all applicable methods.

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

config.jsp:

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

web.xml:

```
1)  <web-app>
2)   <servlet>
```

```
3)      <servlet-name>DemoJSP</servlet-name>
4)      <jsp-file>/config.jsp</jsp-file>
5)      <init-param>
6)        <param-name>hotTopic</param-name>
7)        <param-value>UttarPradesh</param-value>
8)      </init-param>
9)    </servlet>
10) <servlet-mapping>
11)    <servlet-name>DemoJSP</servlet-name>
12)    <url-pattern>/test</url-pattern>
13) </servlet-mapping>
14) </web-app>
```

http://localhost:7777/jsp1/test

The Logical Name: DemoJSP
The Init Param value is : UttarPradesh

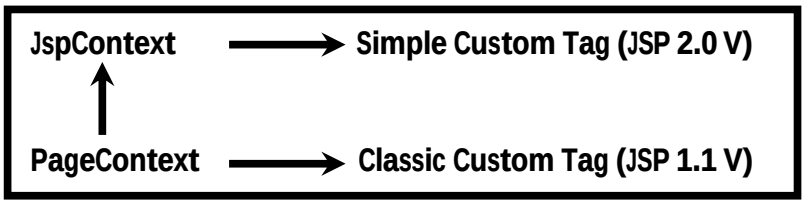
http://localhost:7777/jsp1/config.jsp
The Logical Name: jsp
The Init Param value is : null

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

5) pageContext implicit object:

The pageContext implicit object is of type javax.servlet.jsp.PageContext.

It is abstract class and web server vendor is responsible to provide implementation.
PageContext is the child class of JspContext.



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

1) **To get all other implicit objects**
2) **To Perform Request Dispatching activities**
3) **To perform attribute management in any scope**

a) Getting JSP Implicit objects from pageContext:

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

PageContext class contains the following methods for this.

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

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

b) Request Dispatching by using pageContext:

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

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

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

Eg:

first.jsp:

```
1) <h1>Hello This is First JSP</h1>
2) <%
3)   pageContext.forward("second.jsp");
4) %>
```

second.jsp:

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

forward:

o/p: Hello This is Second JSP

include:

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

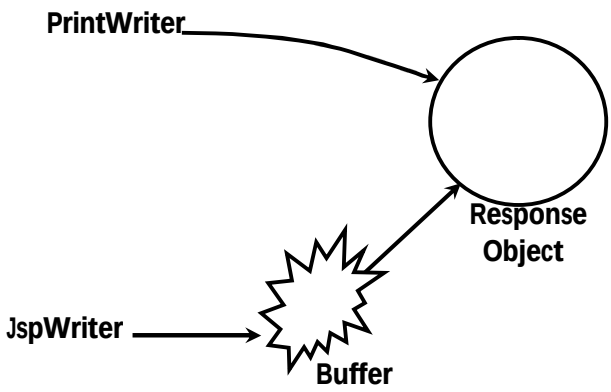
c) Attribute Management in any scope:

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

6) out implicit object:

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

JspWriter = PrintWriter+Buffer



test.jsp:

```
1)  <%@ page import="java.io.*" %>
2)  <%
3)    PrintWriter pw=response.getWriter();
4)    out.print("<h1>Mallika</h1>");
5)    pw.print("<h1>Sunny</h1>");
6)    out.print("<h1>Mallika</h1>");
7)    pw.print("<h1>Sunny</h1>");
8)   out.print("<h1>Mallika</h1>");
9)    pw.print("<h1>Sunny</h1>");
10)  %>
```

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

o/p:

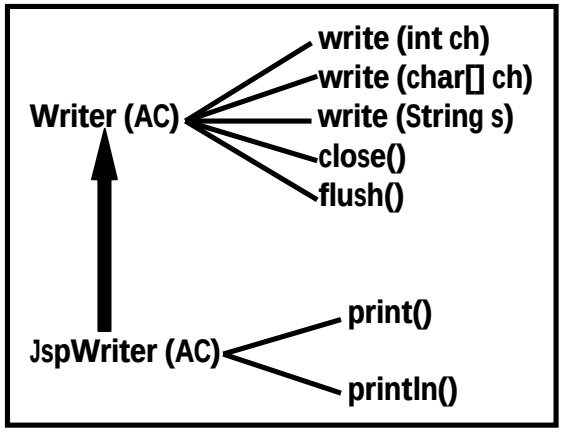
Sunny
Sunny
Sunny

Mallika
Mallika
Mallika

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

Note:

- JspWriter is the child class of Writer(AC).Hence all methods of Writer class are by default available to the JspWriter through inheritance.
- In addition to these methods, JspWriter contains its own set of print() and println() to add any type of Data to the response.



test.jsp:

```
1)  <h1>
2)  <%
3)    out.print("The PI value is :");
4)    out.print(3.14);
5)    out.print("This is exactly ");
6)    out.print(true);
7)  %>
8)  </h1>
```

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

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

7) Page Implicit Object:

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

Note:

- Parent reference can be used to hold child class object. This property is called polymorphism. But by using parent reference, we cannot call child specific methods and we can call only methods present in Parent class.

- Object page=this;

- page variable is declared with Object type. Hence on page variable we can call only methods available in Object class and we cannot call servlet specific methods. If we are trying to call then we will get compile time error.

- If we perform type casting then we can call servlet specific methods.

Q. Which of the following are valid?

1. <%= page.getServletInfo() %> ✕
2. <%= this.getServletInfo() %> ✔
3. <%= getServletInfo() %> ✔
4. <%= ((HttpServlet)page).getServletInfo() %> ✔

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

8) Configuring Error Pages in the JSP (Exception Implicit Object):

- It is not recommended to display java specific error information to the end user directly. We have to convert java specific error information into end user understandable form. For this we have to configure error pages.

- We can configure error pages in JSPs by using the following 2 approaches

    1) Declarative Approach
    2) Programmatic Approach

1) Declarative Approach:

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

```
1) <web-app>
2) <error-page>
```

```
3)    <exception-type>java.lang.ArithmeticException</exception-type>
4)    <location>/error.jsp</location>
5)  </error-page>
6)  <error-page>
7)    <error-code>404</error-code>
8)    <location>/error404.jsp</location>
9)  </error-page>
10) </web-app>
```

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

2) Programmatic Approach:

We can configure error page for a particular jsp by using errorPage attribute of page directive.

demo.jsp:

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

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

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

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

error.jsp:

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

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

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

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

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

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

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

Eg 1:

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

✔

Eg 2:

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

✘

Eg 3:

```
Session Id: <%= session.getId() %>
```

✔

2. Among all JSP implicit objects except session and exception, all remaining objects always available in every JSP. We cannot make these objects unavailable.

session object by default available in every JSP. But we can make it unavailable by using page directive as follows..

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

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

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

JSP Scopes

In Servlets we have the following 3 scopes for storing information in the form of attributes...

1. Request Scope
2. Session Scope
3. Application Scope

In addition to these 3 scopes in JSPs, we have page Scope also...

1. Request Scope:

In Servlets this scope is maintained by ServletRequest object but in JSPs, it is maintained by request implicit object.

The information stored in request scope is available for all the components which are processing that request.

Request scope will start at the time of request object creation(i.e. just before calling service() method) and ends at the time of request object destruction(i.e. just after completing service() method).

Once request processing completed, request scoped attributes will be destroyed.

We can perform attribute management in request scope by using the following methods...

```
public void setAttribute(String name,Object value)
public Object getAttribute(String name)
public void removeAttribute(String name)
public Enumeration getAttributeNames()
```

2. Session Scope:

In Servlets session scope is maintained by HttpSession object, but in JSPs session scope is maintained by session implicit object.

Session scope will be started at the time of session object creation and ends once session expires either by logout mechanism(invalidate() method) or by timeout mechanism.

The information stored in session scope is available for all components which are participating in that session.

HttpSession contains the following methods for attribute management in session scope

```
public void setAttribute(String name,Object value)
public Object getAttribute(String name)
public void removeAttribute(String name)
public Enumeration getAttributeNames()
```

3. Application Scope:

In Servlets this scope is maintained by ServletContext object, but in JSPs this scope is maintained by application implicit object.

The information stored in application scope is available for all components of the web application.

Application scope will be started at the time of ServletContext object creation (i.e. at the time of application deployment or at server startup) and will be ends at the time of context object destruction(ie at the time of application undeployment or server shutdown)

ServletContext interface defines the following methods for attribute management in application scope.

```
public void setAttribute(String name,Object value)
public Object getAttribute(String name)
public void removeAttribute(String name)
public Enumeration getAttributeNames()
```

Q. Write a JSP to print hit count of the application?

hitcount.jsp:

```
1)  <%
2)    Integer count=(Integer)application.getAttribute("hitcount");
3)    if(count==null)
4)    {
5)       count=1;
6)    }
7)    else
8)    {
9)       count++;
10)   }
11)   application.setAttribute("hitcount",count);
12) %>
13) <h1>Hit count of the application is:<%= count %></h1>
```

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

Q. Write a JSP to print number of users login into our application?

sessioncount.jsp:

```
1)  <%
2)    Integer count=(Integer)application.getAttribute("usercount");
3)    if(session.isNew())
4)    {
```

```
5)      if(count==null)
6)      {
7)        count=1;
8)      }
9)      else
10)     {
11)       count++;
12)     }
13)     application.setAttribute("usercount",count);
14)   }
15) %>
16) <h1>The number of users login into our application:<%= count %></h1>
```

Q. Write a JSP to display the number of requests in the current session?

sessionreqcount.jsp:

```
1)  <%
2)    Integer count=(Integer)session.getAttribute("sessionreqcount");
3)    if(count==null)
4)    {
5)      count=1;
6)    }
7)    else
8)    {
9)      count++;
10)   }
11)   session.setAttribute("sessionreqcount",count);
12) %>
13) <h1>The number of requests in the current session: <%= count %> </h1>
```

4. Page Scope:

This scope is applicable only for JSPs but not for Servlets.

This scope is maintained by pageContext implicit object.

The information stored in Page Scope is available only in the current JSP and not available for other JSPs.Hence it is most restricted scope.

Page Scope is the most commonly used scope in custom tags to share information between tag handler classes.

PageContext class defines the following methods for attribute management in page scope.

1. public void setAttribute(String name,Object value)
2. public void removeAttribute(String name)
3. public Object getAttribute(String name)

Extra methods of PageContext class to perform attribute management in any scope:

PageContext class defines the following methods to perform attribute management in any scope...

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

The allowed scopes are:

PageContext.PAGE_SCOPE ➔ 1
PageContext.REQUEST_SCOPE ➔ 2
PageContext.SESSION_SCOPE ➔ 3
PageContext.APPLICATION_SCOPE ➔ 4

2. public Object getAttribute(String name,int scope)

3. public void removeAttribute(String name,int scope)

Q.Which of the following is valid way to store an attribute in Page Scope?

1. page.setAttribute("durga","java"); ✕
2. pageContext.setAttribute("durga","java"); ✔
3. pageContext.setAttribute("durga","java",4); ✕
4. pageContext.setAttribute("durga","java",1); ✔
5. pageContext.setAttribute("durga","java",PageContext.PAGE_SCOPE); ✔

PageContext class defines the following extra methods also..

1. public Enumeration getAttributeNamesInScope(int scope)
2. public Object findAttribute(String name)

First it will searches in page scope and if the attribute present then returns its value. If it is not available then it will search in request, session and then application scopes respectively.
This method acts as all rounder.

Q. What is the difference b/w getAttribute(String name) and findAttribute(String name)?

Q. What is the difference b/w getAttribute(String name) and getAttribute(String name,int scope)?

Demo Program to demonstrate JSP Scopes:

first.jsp:

```
1)  <%
2)  pageContext.setAttribute("p","page");
3)  request.setAttribute("r","request");
4)  session.setAttribute("s","session");
5)  application.setAttribute("a","application");
6)  pageContext.forward("second.jsp");
```

```
7)  %>
```

second.jsp:

```
1)  Page Scope Attribute:<%= pageContext.getAttribute("p") %><br/>
2)  Request Scope Attribute:<%= pageContext.getAttribute("r",2) %><br/>
3)  Session Scope Attribute:<%= pageContext.getAttribute("s",3) %><br/>
4)  Application Scope Attribute:<%= pageContext.getAttribute("a",4) %>
```

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

Demo Program to demonstrate findAttribute()method:

test.jsp

```
1)  <%
2)  pageContext.setAttribute("a","page");
3)  request.setAttribute("a","request");
4)  session.setAttribute("a","session");
5)  application.setAttribute("a","application");
6)  %>
7)  <h1>Find Attribute value :<%= pageContext.findAttribute("a") %><h1>
```

JSP Document

There are 2 Types of syntaxes are possible to write JSPs.

1) JSP Standard Syntax
2) XML Based Syntax

If we are developing JSPs by using JSP Standard syntax, such type of JSPs are called JSP Pages.
If we are developing JSPs by using XML Based syntax, such type of JSPs are called JSP Documents.

1) Scriptlet:

- Standard:
```
<%
    Java Code
%>
```

- XML Based Syntax:
```
<jsp:scriptlet>
    Java Code
</jsp:scriptlet>
```

2) Expression:

- Standard:
```
<%= 2+3 %>
```

- XML Based Syntax:
```
<jsp:expression>
    2+3
</jsp:expression>
```

3) Declaration:

- Standard:
```
<%!
    Java Declarations
%>
```

- XML Based Syntax:
```
<jsp:declaration>
    Java Declarations
</jsp:declaration>
```

4) Directives:

1) <%@ page import="java.util.*" %>
   In XML: <jsp:directive.page import="java.util.*" />

2) <%@ include file="header.jsp" %>
   In XML: <jsp:directive.include file="header.jsp" />

3) <%@ taglib prefix="mine" uri="www.durgasoft.com" %>
   In XML: <jsp:directive.taglib prefix="mine" uri="www.durgasoft.com" /> ➔ Invalid

For the taglib directive there is no equivalent syntax in XML.We can provide equivalent syntax indirectly by using <jsp:root> tag.

```
<jsp:root xmlns:mine="www.durgasoft.com" version="2.3">
        ....
</jsp:root>
```

5) Standard Actions:

There is no difference in standard actions representation b/w standard and xml based syntax.

Eg: <jsp:useBean id="c" class="pack1.CustomerBean" />

6) Comments:

JSP specification does not provide any specific syntax for writing comments.Hence we can use normal xml comments syntax.

<!-- this xml comment -->

7) Template Text:

JSP Standard syntax does not provide any specific way to write template text. But xml based syntax provides a standard way to write template text.

<jsp:text> Any template text</jsp:text>

Comparison b/w JSP Standard syntax and XML Based Syntax

Element	Standard Syntax	XML Based Syntax
Scriptlet	<% Any Java Code %>	<jsp:scriptlet> Any Java Code </jsp:scriptlet>
Expression	<%= X %>	<jsp:expression> X </jsp:expression>
Declaration	<%! Any Java Declarations %>	<jsp:declaration> Any Java Declaration </jsp:declaration>
Directives	page: <%@ page import="..."%>	<jsp:directive.page import="java.util.*"/>
	include <%@ include file="s.jsp"%>	<jsp:directive.include file= "second.jsp"/>
	taglib <%@ taglib prefix="mine" uri="www.durgasoft.com"%>	No Equivalent Tag. But we can fulfill by using <jsp:root> Tag
Standard Actions	<jsp:forward page="second.jsp"/>	No difference <jsp:forward page="second.jsp"/>
Comments	<%-- This is JSP Comment --%>	<!-- This is XML Comment --%>
Template Text	No specific Syntax	<jsp:text> Any Template Text </jsp:text>

How web container identifies JSP Document:

1. Save the JSP File with .jspx extension

2. Enclose total JSP in <jsp:root> tag.

3. use <jsp-config> element in web.xml

```
1)  <web-app>
2)    <jsp-config>
3)      <jsp-property-group>
4)        <url-pattern>*.jspx</url-pattern>
5)      <jsp-xml>true</jsp-xml>
6)      </jsp-property-group>
7)    </jsp-config>
8)  </web-app>
```

Note: From Servlet 2.4 Version onwards Web container can identify JSP Document automatically. No special arrangement is required.

http://localhost:7777/jsp2/jspdoc1.jsp

Write a JSP Document to print hit count of the JSP.

JSP Page:

```
1)  <%!
2)    int count=0;
3)  %>
4)  <%
5)    count++;
6)  %>
7)  The Hit count is:
8)  <%= count %>
```

JSP Document:

```
1)  <jsp:declaration> int count=0;</jsp:declaration>
2)  <jsp:scriptlet>count++; </jsp:scriptlet>
3)  <jsp:text> The Hit count is(XML Based):</jsp:text>
4)  <jsp:expression>count</jsp:expression>
```

Note:

- It is not recommended to use both standard and xml based syntaxes simultaneously.
- The main advantage of XML Based syntax is ,we can use xml editors like XML Spy for writing and debugging JSPs very easily.

Note: In XML Based syntax all tags,attributes are case sensitive. Attribute values must be enclosed either in single quotes or in double quotes.

Q. Which of the following is the valid way of importing java.util package in the JSP?

<jsp:page import="java.util.*" /> ✗
<jsp:directive page import="java.util.*" /> ✗
<jsp:page.directive import="java.util.*" /> ✗
<jsp:directive.page import="java.util.*" /> ✔

Unit 2: JSP Standard Actions

Objective:

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

1. <jsp:useBean> with attributes id, type, class and scope
2.<jsp:getProperty>
3.<jsp:setProperty>

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

test.jsp:

```
1)  <%!
2)    public int squareIt(int x)
3)    {
4)       return x*x;
5)    }
6)  %>
7)  <h1>
8)  The Square of 4 is :<%= squareIt(4) %></br>
9)  The Square of 5 is :<%= squareIt(5) %></br>
10) </h1>
```

This approach has several serious disadvantages...

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

2. This approach does not promote reusability.

3. It reduces readability of the code also

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

CalculatorBean.java:

```
1)  package pack1;
2)  public class CalculatorBean
3)  {
4)     public int squareIt(int i)
5)     {
6)        return i * i;
7)     }
```

```
8)  }
```

test.jsp:

```
1)  <jsp:useBean id="c"  class="pack1.CalculatorBean" />
2)  <h1>The square of 4 is :<%= c.squareIt(4) %> <br>
3)  <h1>The square of 5 is :<%= c.squareIt(5) %> <br>
```

```
beanex1
  |-test.jsp
  |-WEB-INF
    |-classes
      |-pack1
        |-CalculatorBean.class
```

The advantages of this approach are:

1. Separation of Presentation and Business Logic

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

2. Separation of Responsibilities

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

3. It promotes reusability of the code

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

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

Java Bean

Java Bean is a simple java class.

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

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

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

1. Bean Related Tags : <jsp:useBean>

We can use this tag to make bean object available to the JSP.
There are 2 forms of <jsp:useBean>

1. without body:

<jsp:useBean id="c" class="pack1.CalculatorBean" scope="request" />

2. with body:

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

The main objective of body is to perform initialization for newly created bean object..

If the bean object is already available then <jsp:useBean> won't create any new Object and it will use existing object only. At this time body won't be executed.

<jsp:useBean id="c" class="pack1.CalculatorBean" scope="request" />

Attributes of <jsp:useBean>:

<jsp:useBean> can accept the following 5 attributes

1) id
2) class
3) type
4) scope
5) beanName

1. id:

This attribute represents the name of the reference variable of the Bean object.
By using this id only we can access bean in rest of the JSP.

The id attribute is mandatory.

Eg: <jsp:useBean id="p" class="Student" type="Person" />

The equivalent java code is : Person p = new Student();

2. class:

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

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

Bean class should compulsory contains public no-arg constructor. Otherwise we will get Instantiation problems.

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

3. type:

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

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

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

4. scope:

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

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

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

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

Eg: <jsp:useBean id="c" class="pack1.CalculatorBean" scope="request" />

The equivalent java code is:

```
1)  CalculatorBean c=null;
2)  c=(CalculatorBean)pageContext.getAttribute("c",2);
3)  if(c==null)
4)  {
5)    c = new CalculatorBean();
6)    pageContext.setAttribute("c",c,2);
7)  }
```

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

Eg:

```
1)  <%@ page session="false" %>
2)  <jsp:useBean id="c"  class="pack1.CalculatorBean" scope="session"  />
```

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

5. beanName:

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

Various possible combinations:

1. id attribute is mandatory
2. scope attribute is optional and default scope is page scope
3.The attributes class, type, beanName can be used in the following combinations:

a) class
b) type
c) class , type
d) type, beanName (beanName always associated with type but not class)

4. If class attribute is used, whether we are using type attribute or not, it should be concrete class & should contain public no-arg constructor.

Eg:

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

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

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

Eg:

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

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

Q. Consider the class

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

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

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

2. Bean Related Tags : <jsp: getProperty >

We can use this standard action to get properties of the bean object.

Eg:

```
<%
  CustomerBean c = new CustomerBean();
  out.println(c.getName());==>getting and printing
%>
```

equivalent code is

```
<jsp:useBean id="c"  class="CustomerBean" />
<jsp:getProperty  name="c" property="name" />
```

<jsp:getProperty> contains the following 2 attributes

1. name:

The name of bean instance from which required property has to get.
It is exactly equal to id attribute of <jsp:useBean>

2. property:

- The name of java bean property which has to retrieve.
- Both attributes are mandatory.

Note:

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

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

test.jsp:

```
1)  <jsp:useBean id="c" class="pack1.CustomerBean" />
2)  <h1>
3)  Customer Name:<jsp:getProperty name="c" property="name"/><br>
4)  Customer Mail:<jsp:getProperty name="c" property="mail"/>
5)  </h1>
```

CustomerBean.java:

```
1)  package pack1;
2)  public class CustomerBean
3)  {
4)     private String name="durga";
5)     private String mail="durgaocjp@gmail.com";
6)     public String  getName()
7)     {
8)        return name;
9)     }
10)    public String  getMail()
11)    {
12)       return mail;
13)    }
14) }
```

```
beanex2
  |-test.jsp
  |-WEB-INF
    |-classes
      |-pack1
        |-CustomerBean.class
```

3. Bean Related Tags : <jsp: setProperty >

This standard action can be used to set properties of bean object.
We can use <jsp:setProperty> in the following forms:

```
<jsp:useBean id="c" class="CustomerBean" />
```

1.<jsp:setProperty name="c" property="name" value="durga"/>

```
c.setName("durga");
```

2. <jsp:setProperty name="c" property="name" param="uname"/>

```
c.setName(req.getParameter("uname"));
```

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

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

<jsp:setProperty name="c" property="name" />

c.setName(req.getParameter("name"));

4. <jsp:setProperty name="c" property="*" />

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

Attributes of <jsp:setProperty>:

1. name:

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

2. property:

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

3. value:

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

4. param:

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

Demo Program-1 for <jsp:useBean> ,<jsp:getProperty> and <jsp:setProperty>:

login.html:

```
1)  <html>
2)  <body><h1>
3)  <form action="test.jsp">
4)  Name : <input type="text" name="name"><br>
5)  Mail : <input type="text" name="mail"><br>
6)  Age : <input type="text" name="age"><br>
7)  <input type=submit >
8)  </form></h1>
9)  </body>
10) </html>
```

test.jsp:

```
1)  <jsp:useBean id="c" class="pack1.CustomerBean" />
2)  <jsp:setProperty name="c" property="*" />
3)  <h1>Plz confirm your provided values...<br>
4)  Customer Name: <jsp:getProperty name="c" property="name" /><br>
5)  Customer Mail: <jsp:getProperty name="c" property="mail" /><br>
6)  Customer Age: <jsp:getProperty name="c" property="age" /> </h1>
```

CustomerBean.java:

```
1)  package pack1;
2)  public class CustomerBean
3)  {
4)     private String name;
5)     private String mail;
6)     private int age;
7)     public String  getName()
8)     {
9)        return name;
10)    }
11)    public String  getMail()
12)    {
13)       return mail;
14)    }
15)    public int   getAge()
16)    {
17)       return age;
18)    }
19)    public void setName(String name)
20)    {
21)       this.name = name;
22)    }
23)    public void setMail(String mail)
24)    {
25)       this.mail= mail;
26)    }
27)    public void setAge(int age)
28)    {
29)       this.age = age;
30)    }
31) }
```

```
beanex3
  |-login.html
  |-test.jsp
  |-WEB-INF
    |-classes
      |-pack1
```

|-CustomerBean.class

Demo Program-2 for <jsp:useBean> ,<jsp:getProperty> and <jsp:setProperty>:

login.html:

```
1)  <html>
2)  <body><h1>
3)  <form action="test.jsp">
4)  Name : <input type="text" name="name"><br>
5)  Mail : <input type="text" name="mail"><br>
6)  Age : <input type="text" name="age"><br>
7)  <input type=submit >
8)  </form></h1>
9)  </body>
10) </html>
```

test.jsp:

```
1)  <jsp:useBean id="c1" class="pack1.CustomerBean" >
2)      <jsp:setProperty name="c1" property="*"  />
3)  </jsp:useBean>
4)
5)  <h1>Please check your values and confirm<hr>
6)  Name:<jsp:getProperty name ="c1" property="name" /><br>
7)  Mail:<jsp:getProperty name ="c1" property="mail" /><br>
8)  Age: <jsp:getProperty name ="c1" property="age" /><br>
9)  </h1>
```

CustomerBean.java:

```
1)  package pack1;
2)  public class CustomerBean
3)  {
4)     private String name;
5)     private String mail;
6)     private int age;
7)     public String  getName()
8)     {
9)        return name;
10)    }
11)    public String  getMail()
12)    {
13)       return mail;
14)    }
15)    public int   getAge()
16)    {
17)       return age;
18)    }
```

```
19)    public void setName(String name)
20)    {
21)       this.name = name;
22)    }
23)    public void setMail(String mail)
24)    {
25)       this.mail= mail;
26)    }
27)    public void setAge(int age)
28)    {
29)       this.age = age;
30)    }
31) }
```

```
beanex4
  |-login.html
  |-test.jsp
  |-WEB-INF
    |-classes
      |-pack1
        |-CustomerBean.class
```

Note:

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

Developing Reusable Web Components

We can develop reusable web components by using the following standard actions.

1.<jsp:include>
2.<jsp:forward>
3.<jsp:param>

1.<jsp:include>:

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

- The response of header.jsp will be included in the current page response at request processing time. Hence it is dynamic include.
- This is the tag representation of pageContext.include()
- This standard action contains the following 2 attributes.

1. page:

represents the name of the included page. It is mandatory attribute.

2. flush:

It specifies whether the response will be flushed before inclusion or not.
It is optional attribute and default value is false.

Demo Program:

header.jsp

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

footer.jsp:

```
<h1>Ph:8096969696,040-64512786</h1>
```

main.jsp:

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

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

1.include directive:

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

2.include action:

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

3.By using pageContext implicit object:

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

4.By using RequestDispatcher:

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

2. <jsp:forward>:

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

syntax:

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

first.jsp:

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

second.jsp:

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

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

1. forward action:

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

2. By using pageContext implicit object:

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

3. By using RequestDispatcher:

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

3. <jsp:param>

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

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

<jsp:param> contains the following 2 mandatory attributes

1. name:

It represents the name of the parameter

2. value:

It represents the value of the parameter

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

first.jsp:

```
1)  <h1>Welcome to DURGASOFT...</h1>
2)  <jsp:include page="second.jsp">
3)     <jsp:param name="c1" value="JAVA" />
4)     <jsp:param name="c2" value="TESTING" />
5)  </jsp:include>
```

second.jsp:

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

Conclusions:

Case-1:

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

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

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

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

Case-2:

<jsp:forward page="/test" /> ✔

<jsp:include page="/test" /> ✔

<%@ include file="/test" %> ✗

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

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

Case-3:

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

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

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

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

1. Unused Standard Actions : <jsp:plugin>

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

2. Unused Standard Actions : <jsp:fallback>

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

3. Unused Standard Actions : <jsp:params>

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

main.jsp:

```
1)  <jsp:plugin
2)     type="applet"
3)     code="MyApplet.class"
4)     codebase="/plugindemo">
5)
6)     <jsp:params>
7)        <jsp:param name="username" value="durga" />
8)     </jsp:params>
9)
10)    <jsp:fallback>
11)       <p>Could not load applet!!!!</p>
12)    </jsp:fallback>
13)
14) </jsp:plugin>
```

MyApplet.class:

```
1)  import java.awt.BorderLayout;
2)  import java.awt.Color;
3)  import java.awt.Font;
4)
5)  import javax.swing.JApplet;
6)  import javax.swing.JLabel;
7)
8)  public class MyApplet extends JApplet {
9)
10)    private JLabel label = new JLabel();
11)
12)    public void init() {
13)       label.setHorizontalAlignment(JLabel.CENTER);
14)       label.setFont(new Font("Arial", Font.BOLD, 20));
15)       label.setForeground(Color.BLUE);
16)
17)       setLayout(new BorderLayout());
18)       add(label, BorderLayout.CENTER);
19)    }
20)
21)    public void start() {
22)       String firstName = getParameter("username");
23)       label.setText("Hello " + firstName);
24)    }
25) }
```

```
plugindemo
  |-main.jsp
  |-MyApplet
```

http://localhost:7777/plugindemo/main.jsp

Summary of JSP Standard Actions

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

Unit 3: Expression Language (EL)

Agenda:

- EL Introduction
- EL Implicit objects
- EL Operators
- EL Functions
- EL introduced in JSP 2.0V.

EL Introduction:

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

Eg1:

To print the value of request parameter "user"

```
Standard: <%= request.getParameter("user") %>
EL:       ${param.user}
```

Eg2:

To print the value of x attribute

```
Standard: <%= pageContext.findAttribute("x") %>
EL:       ${x}
```

Note:

If the specified attribute is not available then standard syntax will print null, but EL syntax will print blank space(i.e. won't print anything)

EL suppresses null

EL syntax:

If we are using any variable in EL syntax, compulsory it should be some attribute in some scope.

```
${x}
```

EL Implicit Objects:

EL contains 11 implicit objects. The power of EL is just because of these implicit objects only.

The following is the list of all EL Implicit objects

1) pageScope
2) requestScope
3) sessionScope
4) applicationScope

} Map of Scoped Attributes

5) param
6) paramValues

} Map of Request Parameters

7) header
8) headerValues

} Map of Request Headers

9) cookie → Map of Cookies

10) initParam → A Map of Context Initialization Parameters (But not Servlet Initialization Parameters)

11) pageContext

1. pageScope,requestScope,sessionScope and applicationScope

We can use these implicit objects to retrieve attributes of a particular scope

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

Eg1:

To get the value of session scoped attribute x

```
${sessionScope.x}
```

Eg2:

To get the value of request scoped attribute x

```
${requestScope.x}
```

Eg3:

```
${x}
```

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

test.jsp:

```
1)  <%
2)    pageContext.setAttribute("p","Pavan");
3)    pageContext.setAttribute("p","ravi",2);
4)    pageContext.setAttribute("p","sunny",3);
5)    pageContext.setAttribute("p","anushka",4);
6)  %>
7)  <h1>${pageScope.p}<br>
8)  ${requestScope.p}<br>
9)  ${sessionScope.p}<br>
10) ${applicationScope.p}<br>
11) ${p}<br>
```

Output:

Pavan
ravi
sunny
anushka
Pavan

2. param and paramValues

We can use these implicit objects to retrieve request form parameters.

param → getParameter()
paramValues → getParameterValues()

${param.x}

It prints the value of form parameter x

If the parameter associated with multiple values then we will get only first value.

If the specified parameter not available then we will get blank space

${paramValues.x}

Internally it returns String[] which contains all values associated x. as we are printing that String[], internally toString() method will be called, which prints result in the following form

[Ljava.lang.String;@1270b73

${paramValues.x[0]} → It prints first value of x

${paramValues.x[1]} ➔ It prints second value of x
${paramValues.x[100]} ➔ prints blank space as EL suppresses ArrayIndexOutOfBoundsException

Demo Program:

details.jsp:

```
1) <form action="el3.jsp" >
2) Enter name: <input type=text name=uname><br>
3) Enter EmpId:<input type=text name=eid><br>
4) Enter Food1:<input type=text name=food><br>
5) Enter Food2:<input type=text name=food><br>
6) <input type=submit>
7) </form>
```

el3.jsp:

```
1) <h1>
2) Name:${param.uname}<br>
3) Emp Id: ${param.eid}<br>
4) Food: ${param.food}<br>
5) Food[]:${paramValues.food}<br>
6) Food1: ${paramValues.food[0]}<br>
7) Food2: ${paramValues.food[1]}<br>
8) Food100: ${paramValues.food[100]}<br>
9) </h1>
```

3. header and headerValues:

These are exactly same as param and paramValues except that these are for retrieving request headers.

header ➔ request.getHeader()
headerValues ➔ request.getHeaders()

Eg:
<h1>${header.accept}
<h1>${header.host}
<h1>${headerValues.host[0]}

4. cookie implicit object :

By using this implicit object we can get cookies associated with the request.
cookie ➔ request.getCookies()

<h1>${cookie.JSESSIONID.name}
${cookie.JSESSIONID.value}

5. initParam implicit object:

We can use initParam implicit object to access context parameters declared in web.xml

initParam ➔ context.getInitParameter()

Demo Program:

web.xml:

```
1)  <web-app>
2)    <context-param>
3)      <param-name>user</param-name>
4)      <param-value>scott</param-value>
5)    </context-param>
6)    <context-param>
7)      <param-name>pwd</param-name>
8)      <param-value>tiger</param-value>
9)    </context-param>
10) </web-app>
```

el6.jsp:

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

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

6. pageContext implicit object:

This is the only common implicit object between JSP and EL.

This is the only EL implicit object which is a non-map object.

By using this implicit object we can access all other JSP implicit objects in EL.

Eg:
${request.method} ➔ invalid
${pageContext.request.method} ➔ valid
${session.id} ➔ invalid
${pageContext.session.id} ➔ valid

In EL, request and session implicit objects are not available.

Note: El handles null and ArrayIndexOutOfBoundsException very nicely and prints blank space

EL Operators

EL contains its own specific operators. The following is the list of all possible operators.

1. Property Access operator(.)
2. Collection Access Operator([])
3. Arithmetic Operators
4. Relational Operators
5. Logical Operators

1. Property Access operator(.)

Syntax: ${leftvariable.rightvariable}

Left Variable ➔ Map OR Bean
Right Variable ➔ Map Key OR Property
Should be Valid Java Identifier

Eg:

${customer.name} ➔ valid
${pageContext.request.method} ➔ valid
${initParam.user} ➔ valid
${customer.1234} ➔ invalid

2. Collection access operator([]):

Syntax: ${leftvariable[rightvariable]}

map ➔ key
bean ➔ property
} Should be enclosed either in Single OR Double Quotes

array ➔ index
list ➔ index
} Quotes are Optional

key or property should be enclosed in either single quotes or double quotes. For index quotes are optional.

Case-1:working with Map:

${initParam["user"]}
${initParam['user']}
${initParam[user]} blank space

If we are not keeping quotes EL assumes that the key is an attribute stored in some scope. If that attribute is not available then we will get blank space as output.

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

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

Case-2: working with Bean:

${customer['name']}
${customer["name"]}
${customer[name]} blank space as o/p

case-3: Working with arrays:

```
1)  <%
2)    String[] s ={"AAA","BB","CCCC"};
3)    pageContext.setAttribute("s",s);
4)  %>
5)  <h1>
6)  ${s[0]}
7)  ${s["1"]}
8)  ${s['2']}
9)  ${s[3]}==>blank space
```

Case-4: working with List objects:

```
1)  <%@ page import="java.util.*" %>
2)  <%
3)    ArrayList l = new ArrayList();
4)    l.add("AAA");
5)    l.add("BBB");
6)    l.add("CCC");
7)    l.add("DDD");
8)    pageContext.setAttribute("list",l);
9)    pageContext.setAttribute("index",3);
10) %>
11) <h1>
12) ${list[0]}<br/>
13) ${list["1"]}<br/>
14) ${list['2']}<br/>
15) ${list[index]}
```

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

3.Arithmetic operators:

1.+ operator:

EL does not support overloading and hence there is no string concatenation operator. "+" always acts as addition operator

Eg:

${2+3} ➔ 5
${"abc"+3} ➔ RE: NumberFormatException
${abc+3} ➔ 3
${""+3} ➔ 3
${null+3} ➔ 3

Note:
1. In Arithmetic operators null is treated as "0"
2. Empty String is also treated as "0"

2. - operator:

All rules are exactly same as + operator

${10-3} ➔ 7
${"10"-3} ➔ 7
${"abc"-3} ➔ RE:NFE
${abc-3} ➔ -3

3. * operator:

All rules are exactly same as + operator
${10 *3} ➔ 30

4. / operator or (div)

All rules are exactly same as + operator except that division operator always follows floating point arithmetic

${10 / 2} ➔ 5.0
${10 div 2} ➔ 5.0
${10 / 0} ➔ Infinity
${0 / 0} ➔ NaN

5. % operator (or) mod operator:

In the case of mod operator both floating point and integral arithmetic are possible.

${10%0} ➔ ArithmeticException
${10%3} ➔ 1

Relational operators:

> or gt
< or lt
== or eq
>= or ge
<= or le
!= or ne

${10>3} ➔ true
${abcd==abc} ➔ true

Logical Operators:

There are 3 logical operators
& or && or and
| or || or or
! or not

Eg:
${!true} ➔ false
${true && false} ➔ false

Conditional operator:

${(3<4)?"yes":"no"} ➔ yes

Empty Operator:

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

otherwise returns false.

Eg:
${empty durga} ➔ true
${empty "durga"} ➔ false
${empty null} ➔ true

EL operator precedence:

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

EL Reserved Words:

- true
- false
- null
- empty
- instanceof
- gt
- lt
- ge
- le
- ne
- eq
- and
- or
- mod
- div
- not

EL versus null:

EL behaves very nicely with null.

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

Eg:

${10+null} ➔ 10
${empty null} ➔ true
${!null} ➔ true

EL Functions

The main objective of EL is to separate java code from the JSP. If we are having any business functionality we can separate it to a java class & we can invoke its functionality through EL syntax.

The page designer has to know only function name and tld file uri to get its functionality. Hence EL is almost consider as alternative to custom tags.

Steps to develop EL Function based application:

1. Write a java class with required functionality

2. Write tld file to map java class to JSP

3. Write a taglib directive to make business functionality available to the JSP

4.write EL Function call

1.Writing a java class:

Any java class can acts as repository for EL functions. The only requirement of a method that acts as EL Function is it should be declared as public and static.
Method can take parameters also.
void return type methods are not recommended to use as EL functions even though it is legal.

```
1)  package pack1;
2)  public class DiceRoller
3)  {
4)     public static int rollDice()
5)     {
6)        return (int)((Math.random()*6)+1);
7)     }
8)  }
```

2. Writing tag library descriptor file(tld file):

For EL functions, tld file provides mapping between java class (where functionality is available) & JSP (where functionality is required).

We can configure EL function by using <function> tag in the tld file.

<function> tag contains the following 4 child tags

1. <description>

2. <name>
   By using this name only we can call EL functionality in the JSP.

This name need not be original name of the method

3. <function-class>
The fully qualified name of java class where EL function is defined

4. <function-signature>
The signature of EL function which is defined in java class.
Here we have to provide return type, method-name & argument list

Eg: myfunctions.tld:

```
1)  <taglib version="2.1" >
2)  <tlib-version>1.2</tlib-version>
3)  <uri>DiceFunctions</uri>
4)
5)  <function>
6)     <name>rollIt</name>
7)     <function-class>pack1.DiceRoller</function-class>
8)     <function-signature>int rollDice()</function-signature>
9)  </function>
10)
11) </taglib>
```

3. Write taglib directive:

we can write taglib directive to make EL Functions available to the JSP

<%@ taglib prefix="mime" uri="DiceFunctions" %>

4. Write EL function call:

${ mime:rollIt()}

Demo Program:

test.jsp:

```
1)  <%@ taglib prefix="mine" uri="DiceFunctions" %>
2)  <h1>Your 3-digit Lucky number is:
3)  ${mine:rollIt()}
4)  ${mine:rollIt()}
5)  ${mine:rollIt()}</h1>
```

MyFunctions.tld

```
1)  <taglib version="2.1" >
2)   <tlib-version>1.2</tlib-version>
3)   <uri>DiceFunctions</uri>
```

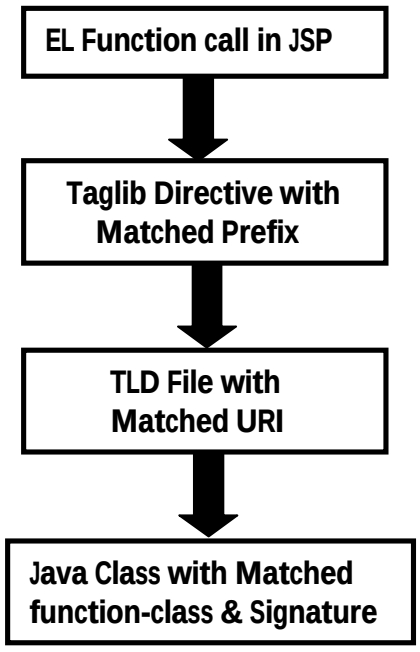
```
4)
5)    <function>
6)       <name>rollIt</name>
7)       <function-class>pack1.DiceRoller</function-class>
8)       <function-signature>int rollDice()</function-signature>
9)    </function>
10)
11) </taglib>
```

DiceRoller.java:

```
9)  package pack1;
10) public class DiceRoller
11) {
12)    public static int rollDice()
13)    {
14)       return (int)((Math.random()*6)+1);
15)    }
16) }
```

```
elApp1
  |-test.jsp
  |-WEB-INF
    |-MyFunctions.tld
    |-classes
      |-pack1
            |-DiceRoller.class
```

Execution Flow of EL Function



1. Whenever JSP engine encounters EL Function call with Prefix and EL Function Name, then it checks for corresponding taglib directive with matched prefix.

2. From taglib directive JSP engine identifies uri and checks for tld file with matched uri.

3. From the TLD file JSP engine identified java class and method signature.JSP engine will check for the matched java class and method signature.

4. JSP Engine executes that method and return its results to JSP.

Demo Program-2:

Write application to invoke Math class sqrt() method as EL Function call with method name m1().
public static double sqrt(double d)

test.jsp:

```
1)  <%@ taglib prefix="mine" uri="MathFunctions" %>
2)
3)  <h1>The Square Root of 4 is:${mine:m1(4)}<br>
4)  <h1>The Square Root of 36 is:${mine:m1(36)}<br>
```

MyTld.tld:

```
1)  <taglib version="2.1">
2)    <tlib-version>1.2</tlib-version>
3)    <uri>MathFunctions</uri>
4)    <function>
5)      <name>m1</name>
6)      <function-class>java.lang.Math</function-class>
7)      <function-signature>double sqrt(double)</function-signature>
8)    </function>
9)  </taglib>
```

```
elApp2
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
```

Unit 4: JSTL (JSP Standard Tag Library)

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

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

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

Total JSTL Library dividing into 5 sub parts.

1.Core Library:

It defines some standard actions to perform programming general stuff like conditions & loops. It can also perform JSP fundamental tasks such as setting attributes, writing output & redirecting the request to other pages.

2. SQL Library:

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

3. Functional Library:

Defines several standard actions which can be used for manipulating String objects.

4.FMT(formating) Library:

Defines several standard actions which can be used for formatting numbers and dates for Internationalization purpose.

5. xml Library:

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

Core Library:

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

1. General Purpose Tags

```
<c:out>
<c:set>
<c:remove>
<c:catch>
```

2. Conditional Tags

<c:if>
<c:choose>
<c:when>
<c:otherwise>

3. Iteration Tags

<c:forEach>
<c:forTokens>

4. URL Related Tags

<c:import>
<c:url>
<c:redirect>
<c:param>

Installing JSTL:

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

1. jstl.jar:

Defines several API classes defined by SUN.

2. standard.jar:

Contains implementation classes provided by vendor.

Note:

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

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

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

General Purpose Tags:

1. <c:out>:

<c:out>

where c is prefix and out is name of tag.

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

form-1:

<c:out value="durga" />
It prints durga to JSP

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

form-2:

If the main value is not available or it evaluates to null, then we can provide default value by using default attribute.

<c:out value="${param.user}" default="Guest" />

test.jsp:

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

http://localhost:7777/jsp1/test.jsp?uname=DURGASOFT

o/p: Hello DURGASOFT

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

o/p: Hello Guest

Note:

<c:out> can accept the following 3 attributes

1.value
2.default
3.escapeXml

2.<c:set>

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

Form-1:

We can use <c:set> to set attributes in any scope

<c:set var="name of the attribute" value="attributevalue" scope="request" />
scope attribute is optional and default scope is page scope.

test.jsp:

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

form-2:

We can set map or bean properties by using <c:set>
We can specify bean or map object by using target attribute and property name by using property attribute.

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

The attributes for the <c:set> are:

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

3. <c:remove>:

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

This tag can take the following 2 attributes

1.var ➔ Name of the Attribute
2.scope ➔ The Scope in which Attribute Present

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

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

test.jsp:

```
1)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2)  <c:set var="x" value="40" scope="page" />
3)  <c:set var="y" value="60" scope="page" />
4)  <c:set var="sum" value="${x+y}" scope="session"  />
5)  <h1>
6)  The result is : <c:out value="${sum}" /><br>
7)  <c:remove var="x" />
8)  <c:remove var="y" />
9)  <c:remove var="sum" />
10) The result is : <c:out value="${sum}" default="1000"/><br>
11) </h1>
```

<c:catch>:

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

```
<c:catch >

 Risky Code

</c:catch>
```

If an exception raised in the risky code, then this tag suppresses that exception and rest of the JSP will be executed normally.

If an exception is raised we can hold that by using var attribute, which is the page scoped attribute.

test.jsp:

```
1)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2)  <h1>
3)    User Name: ${param.uname}<br>
4)    <c:catch var ="e" >
5)          <%
6)            int age = Integer.parseInt(request.getParameter("uage"));
7)          %>
8)      Age:${param.uage}<br>
9)
10)  </c:catch>
11)  <c:if test="${ e != null}">
12)    Oops ...Exception raised :  ${e}<br>
13)  </c:if>
14) User Height: ${param.uheight}
15)
16) </h1>
```

http://localhost:7777/jsp1/catch1.jsp?uname=pavan&uage=47&uheight=5.11

User Name: pavan
Age:47
User Height: 5.11

http://localhost:7777/jsp1/catch1.jsp?uname=pavan&uage=ten&uheight=5.11

User Name: pavan
Oops ...Exception raised : java.lang.NumberFormatException: For input string: "ten"
User Height: 5.11

Summary of General Purpose Tags

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

Conditional Tags

1.<c:if>

We can use to implement core java if statement.
There are 2 forms are available for <c:if>

without body:

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

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

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

test.jsp:

```
1)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2)  <c:set var="x" value="10" scope="request"/>
3)  <c:if test="${x ne '10'}"  var="y" scope="session" />
4)
5)  <h1>
6)  X value is : ${x}<br>
7)  Test Result is : ${y}
8)  </h1>
```

with body:

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

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

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

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

test.jsp:

```
1)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2)  <c:set var="x" value="10" scope="request"/>
3)
4)  <c:if test="${x eq '10'}" >
5)     <h1> x value is  equal to 10</h1>
6)  </c:if>
```

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

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

1. Implementing if-else:

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

```
1)  <c:choose>
2)
3)    <c:when test="test_condition">
4)       Action-1
5)    </c:when>
6)
7)    <c:otherwise>
8)       Action-2
9)    </c:otherwise>
10)
11) </c:choose>
```

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

Implementing switch statement:

```
1)  <c:choose>
2)    <c:when test="test_condition1">
```

```
3)        Action-1
4)     </c:when>
5)     <c:when test="test_condition2">
6)        Action-2
7)     </c:when>
8)     .....
9)     <c:when test="test_conditionN">
10)       Action-N
11)    </c:when>
12)    <c:otherwise>
13)       Default Action
14)    </c:otherwise>
15) </c:choose>
```

1.<c:choose> should compulsory contains at least one <c:when> tag. But <c:otherwise> is optional.

2. Every <c:when> tag implicitly contains break statement. Hence there is no chance of fall through inside switch.

3. We should take <c:otherwise> as last case only.

4.<c:choose> and <c:otherwise> won't take any attributes but <c:when> can take one mandatory attribute test.

switch.jsp:

```
1)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2)  <h1> Select The Number:
3)  <form action="/jsp1/switch.jsp">
4)  <select name="combo" >
5)    <option value="1">1</option>
6)    <option value="2">2</option>
7)    <option value="3">3</option>
8)    <option value="4">4</option>
9)    <option value="5">5</option>
10) <option value="6">6</option>
11) <option value="7">7</option>
12) </select>
13) <input type="submit"/>
14) </form>
15) <c:set var="s" value="${param.combo}"/>
16) Today is :
17) <c:choose>
18)   <c:when test="${s == 1}" >Sunday</c:when>
19)   <c:when test="${s == 2}" >Monday</c:when>
20)   <c:when test="${s == 3}" >Tuesay</c:when>
21)   <c:when test="${s == 4}" >Wednesday</c:when>
22)   <c:when test="${s == 5}" >Thursday</c:when>
23)   <c:otherwise>Select between 1 and 5</c:otherwise>
```

```
24) </c:choose>
25) </h1>
```

Summary of Conditional Tags

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

Iteration Tags:

There are 2 iteration tags are available in JSTL.

1. <c:forEach> ➔ General purpose for loop

2. <c:forTokens> ➔ Specialized for splitting Strings similar to StringTokenizer.

1.<c:forEach>:

form-1:

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

o/p: 10 times

<c:forEach begin="1" end="10" step="2" > 5 times

<c:forEach begin="4" end="0" step="-1" > Invalid b'z step value should not be -ve.

The begin attribute specifies the index where the loop has to start.

end index specifies the index where the loop has to terminate.

This loop internally maintains counter variable which is incremented by step attribute value.

The default value of step attribute is "1" and it is optional attribute.

form-2: <c:forEach> with var attribute:

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

test.jsp:

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

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

```
1)  <c:forEach items="collection object" var="current object">
2)
3)     body
4)
5)  </c:forEach>
```

Items attribute should be either Collection object or array object. This action will iterate over each item in the collection until all elements completion.

We can represent current collection object by using var attribute.

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

test.jsp:

```
1)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2)  <%
3)    String[] s = {"A","B","C","D"};
4)    pageContext.setAttribute("s",s);
5)   %>
6)
7)  <c:forEach items="${s}" var="obj">
8)    <h1>The current Object is : ${obj} </h1>
9)  </c:forEach>
```

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

test.jsp:

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

```
5)   <tr> <td>${hdr.key}</td><td>${hdr.value}</td></tr>
6)  </c:forEach>
7)  </table >
```

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

test.jsp:

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

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

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

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

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

This class contains several methods which are useful during iteration.

1. Object getCurrent()
returns the current item

2. int getIndex()
returns the current index(counter value)

3. int getCount()
returns the number of iterations that have already performed.

4. boolean isFirst()
Returns true if the current Iteration is the First iteration

5. boolean isLast()
Returns true if the current iteration is the last iteration

6. Integer getBegin()
returns start index

7. Integer getEnd()
returns end index

8. Integer getStep()

returns the incremented value

test.jsp:

```
1)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2)
3)  <c:forEach items="durga,pavan,ravi,shiva"  varStatus="status" >
4)
5)   <h1>Is it First Iteration: ${status.first}<br>
6)   The current Object is : ${status.current}<br>
7)   The number of iterations already completed:${status.count}<br>
8)   Is it Last Iteration :${status.last}<br><hr></h1>
9)
10) </c:forEach>
```

<c:forTokens>:

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

test.jsp:

```
1)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2)
3)  <c:forTokens items="durga.pavan.shiva.ravi" delims="." var="x">
4)   <h1>Hello:${x}<br>
5)  </c:forTokens>
```

<c:forTokens> can accept the following extra attributes also.

begin: specifies the index at which iteration should start. The index of first token is zero.

end: specifies the index where iteration should be terminated.

step: counter increment value between iterations

varStatus: to specify the status of iteration

test.jsp:

```
1)  <%@ page isELIgnored="false" %>
2)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3)
4)  <c:forTokens items="one,two,three,four,five,six" delims="," var="x" begin="2" end="5" st
    ep="1">
5)   <h1>Current Token is : ${x}</h1>
6)  </c:forTokens>
```

Note:

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

Summary of Iteration Tags

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

URL Related Tags

1. <c:import>

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

Form-1:

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

Eg:

first.jsp:

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

second.jsp:

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

Form-2:

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

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

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

Form-3:

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

test.jsp:

```
1)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2)
3)  <c:import url="second.jsp" var="x" scope="request" />
4)  ${x}<br>
5)  ${x}<br>
6)  ${x}<br>
7)  ${x}<br>
8)  ${x}<br>
```

Form-4:

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

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

Form-5:

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

first.jsp:

```
1)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2)
3)  <h1>Welcome to DURGASOFT!!!!</h1>
4)  <c:import url="second.jsp" >
5)     <c:param name="c1" value="JAVA"/>
6)     <c:param name="c2" value="PHP"/>
7)  </c:import>
```

second.jsp:

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

2.<c:redirect>:

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

Form-1:

<c:redirect url="second.jsp" />
url can be specified either with Relative path or absolute path

first.jsp:

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

second.jsp:

<h1>Hello this is Second JSP</h1>
If we send the request to first.jsp then second.jsp will provide response through redirection.

Form-2:

We can redirect request to some other web application's resources also.

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

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

Form-3:

While performing redirection we can pass form parameters also to target resource.

first.jsp:

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

second.jsp:

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

3.<c:url>

We can use this standard action to rewrite urls to append session information and form parameters.

Form-1:

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

The encoded url with sessionid will be stored inside x.

test.jsp:

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

Form-2:

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

Form-3:

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

first.jsp:

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

second.jsp

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

Summary of URL related Tags

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

Summary of All Core Library Tags:

1. <c:out> To display expression to the JSP
2. <c:set> to set attributes and bean properties
3. <c:remove> to remove attributes from the specified scope
4. <c:catch> To suppress exception and continue rest of the jsp
5. <c:if> To implement core java if statement
6. <c:choose>,<c:when>,<c:otherwise> To implement if-else and switch statements

9. <c:forEach> To implement general purpose for loop
10. <c:forTokens> For string tokenization purpose

11. <c:import> for dynamic include
12. <c:redirect> for redirection
13. <c:param> to send parameters while importing and redirecting
14. <c:url> to append session information to the url for session management purpose

JSTL SQL Library

It defines several tags for communicating with database.

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

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

The following are important sql library tags

1. <sql:setDataSource>
To create DataSource

2. <sql:query>
To execute select query

3. <sql:update>
To execute non-select query(insert|delete|update)

4. <sql:param>
To provide parameter values in the case of PreparedStatement

5. <sql:dateParam>
To provide date values in the case of PreparedStatement

6. <sql:transaction>
To implement transactions

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

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

test.jsp:

```
1)  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
2)  <%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>
3)
4)  <sql:setDataSource var="ds" driver="oracle.jdbc.OracleDriver"
5)      url="jdbc:oracle:thin:@localhost:1521:XE"
6)      user="scott"  password="tiger"/>
7)
8)  <sql:query dataSource="${ds}" var="result">
9)  SELECT * from Employees
10) </sql:query>
11) <h1>
12) <c:forEach items="${result.rows}" var="row" >
13)  ${row.eno}--${row.ename}--${row.esal}--${row.eaddr}<br>
14) </c:forEach>
15) </h1>
```

~~~~~~

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

~~~~~~

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

~~~~~~

```
1)  <sql:update dataSource="${ds}" var="count">
2)   update employees set esal=7777 where ename='durga'
3)  </sql:update>
4)  <h1>The number of rows updated:${count}</h1>
```

~~~~~~

```
1)  <sql:update dataSource="${ds}" var="count">
2)   update employees set esal=? where eno=?
```

```
   <sql:param value="9999"/>
   <sql:param value="100"/>
  </sql:update>
  <h1>The number of rows updated:${count}</h1>
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
<sql:transaction dataSource="${ds}">

    <sql:update>
     update employees set esalesal=esal-1000 where ename='durga'
    </sql:update>

    <sql:update >
     update employees set esalesal=esal+1000 where ename='sunny'
    </sql:update>

</sql:transaction>
```

JSTL Functional Library

It defines several tags for general String manipulation operations.

To use Functional Library,we have to write the following taglib directive in JSP.

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

The following are important tags in this library:

1.<fn:contains()>

Eg2:

split() method will split the given string according to the specified separator (delimiter)

join() method will join the given tokens into a single string based on separator.

Eg1:

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:set var="s" value="Hello Learning JSTL is Very Easy"/>
<h1>
Length: ${fn:length(s)}<br>
In Upper Case: ${fn:toUpperCase(s)}<br>
In Lower Case: ${fn:toLowerCase(s)}<br>
Sub String from index 6 to 15: ${fn:substring(s,6,15)}<br>
```

```
9)  Is s contains JSTL: ${fn:contains(s,"JSTL")}<br>
10) Is s starts with Hello: ${fn:startsWith(s,"Hello")}<br>
11) Is s ends with Easy: ${fn:endsWith(s,"Easy")}<br>
```

~~~~~~

Eg2:

```
1)  <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
2)  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3)  <c:set var="data" value="sunny,mallika,veena,reshmi,anasuya"/>
4)
5)  <c:set var="s" value="${fn:split(data,',')}"/>
6)  <h1>
7)   Result of split method:<br>
8)  <c:forEach items="${s}" var="s1">
9)   ${s1}<br>
10) </c:forEach>
11)
12) <c:set var="result" value="${fn:join(s,'-')}"/>
13) Result of Joining:  ${result}
```
~~~~~~

Unit 5: Custom Tags

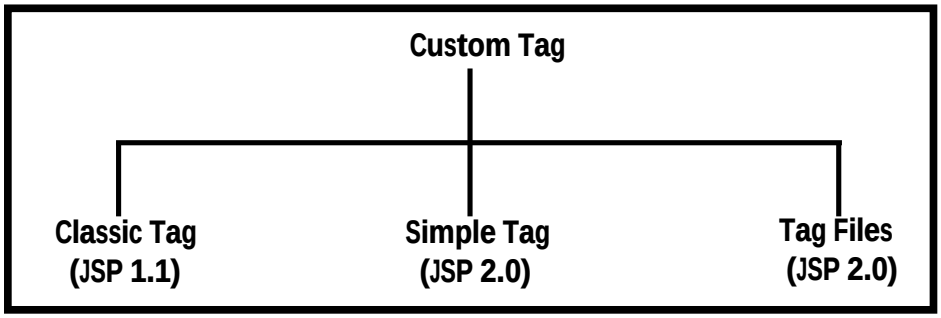
Standard actions, EL and JSTL are not succeeded to separate java code completely from the JSP.

For example our requirement is to provide sports information or movie information etc. There are no standard actions defined for these requirement.

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

All custom tags are divided into 3 categories.

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



Components of Custom Tag Application:

Custom tag application contains the following 3 components

1.Tag Handler class:

It is a simple java class which defines entire required functionality

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

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

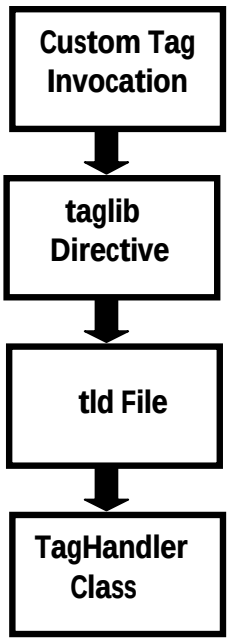
2.TLD file(Tag Library Descriptor):

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

3.taglib directive:

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

Execution Flow of Custom tag application



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

2. From the taglib directive jsp engine identifies the location of the tld file

3. From the tld file JSP engine identifies the corresponding Tag handler class.

4. JSP engine executes tag handler class and provides required functionality to the JSP.

Tag Extension API:

We can build custom tags by using only one package:

javax.servlet.jsp.tagext

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

1. Tag(I):

It is the base interface for all custom tag handlers.

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

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

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

2. IterationTag(I):

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

It contains only one extra method: doAfterBody()

3. BodyTag(I):

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

This interface defines 2 extra methods

setBodyContent() and doInitBody()

4. TagSupport(C):

It implements IterationTag interface and provides default implementation for all its methods.
It acts as base class to develop simple and Iteration tags.

More or less this class acts as adapter class for Tag and IterationTag interfaces.

5. BodyTagSupport(C):

This class implements BodyTag interface and provides default implementation for all its methods.
It is the child class of TagSupport.

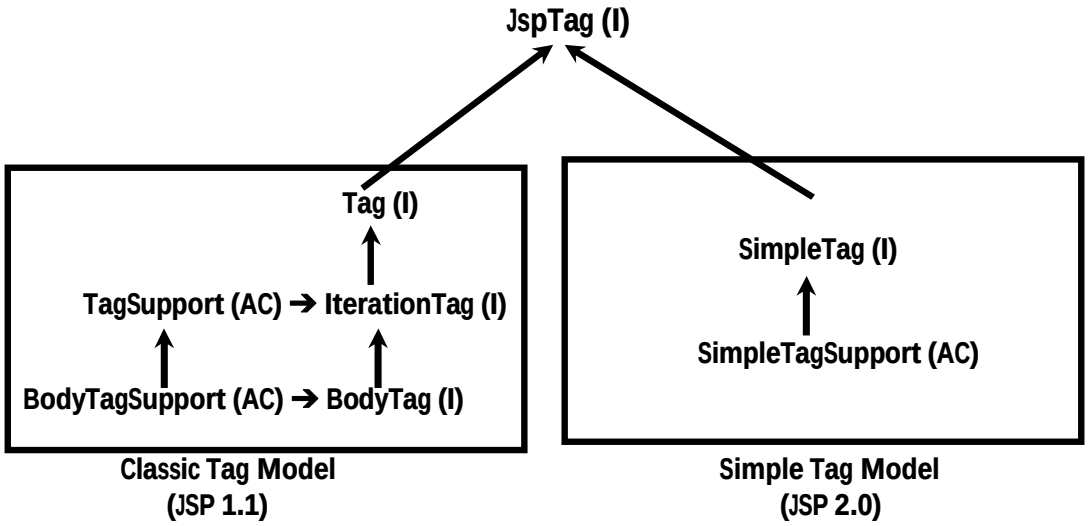
We can use this class as Base class for implementing Customtags that processes tag body.

6.BodyContent(C):

BodyContent object acts as a buffer to hold tag body.

It extends JspWriter.

We have to use BodyContent class only in BodyTag interface and BodyTagSupport class.



JspTag Interface is just for Polymorphism purpose and doesn't contain any Methods.

Implementing Tag(I):

Tag interface defines the following 6 methods:

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

Tag interface defines the following 4 constants

EVAL_BODY_INCLUDE
SKIP_BODY
EVAL_PAGE
SKIP_PAGE

Life Cycle of Tag Handler that implements Tag Interface

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

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

3.JSP Engine calls setPageContext() method to make pageContext object available to Tag Handler class.

```
public void setPageContext(PageContext p)
```

By using pageContext implicit object, Tag Handler class can get all other implicit objects and can get attributes from various scopes.

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

```
public void setParent(Tag t)
```

5.Setting Attributes

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

JSP engine calls setter methods for each attribute.

6. JSP engine calls doStartTag()

```
public int doStartTag() throws JspException
```

We can define entire tag functionality in this method only.

doStartTag() method can return either EVAL_BODY_INCLUDE or SKIP_BODY. If it returns EVAL_BODY_INCLUDE then tag body will be included in the result.

If it returns SKIP_BODY then JSP Engine won't consider tag body.

7.JSP Engine calls doEndTag()

```
public int doEndTag()throws JspException
```

doEndTag() can return either EVAL_PAGE or SKIP_PAGE.

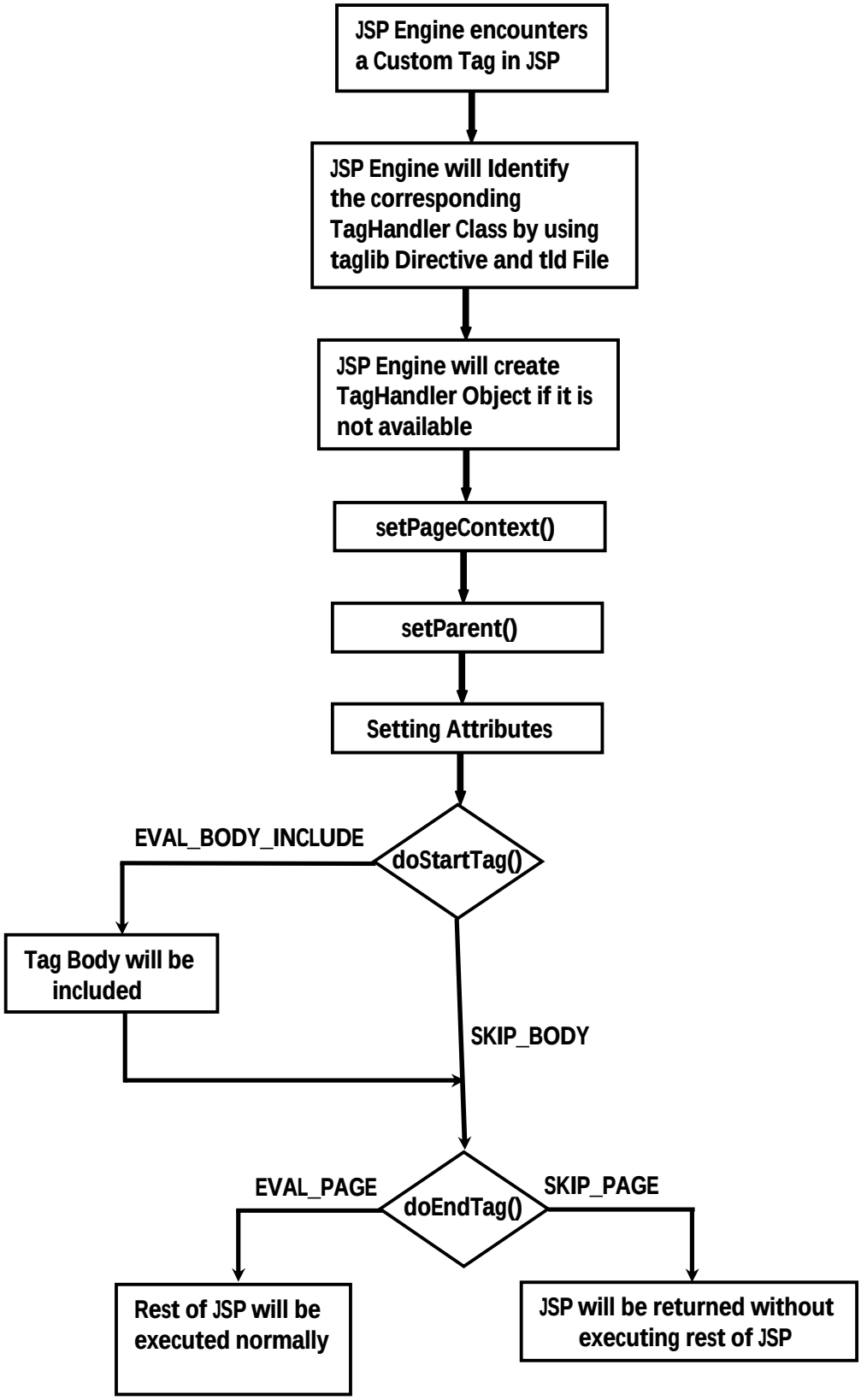
If it returns EVAL_PAGE then rest of the JSP will be executed normally.

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

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

```
public void release()
```

Flowchart for Tag Handler Life Cycle



Demo Program for Custom Tags(cust1)

test.jsp:

```
1)  <%@taglib  prefix="mine" uri="/WEB-INF/MyTld.tld" %>
2)  <h1>Hello this is Demo JSP</h1>
3)
4)  <mine:mytag>
5)    <H1>This is body of the custom tag</H1>
6)  </mine:mytag>
7)
8)  <h1>This is after the custom tag invocation</h1>
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)   <tlib-version>1.2</tlib-version>
3)
4)      <tag>
5)       <name>mytag</name>
6)       <tag-class>tags.MyCustomTag</tag-class>
7)      </tag>
8)
9)  </taglib>
```

MyCustomTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  import java.io.*;
5)  public class MyCustomTag implements Tag
6)  {
7)     private PageContext pageContext;
8)     public void setPageContext(PageContext pageContext)
9)     {
10)       this.pageContext = pageContext;
11)    }
12)    public void setParent(Tag t){   }
13)    public int doStartTag() throws JspException
14)    {
15)       try{
16)       JspWriter out = pageContext.getOut();
17)       out.println("<h1>Hello this is from tag handler</h1>");
18)       }catch(IOException e){}
19)       return EVAL_BODY_INCLUDE;
20)    }
21)    public int doEndTag() throws JspException
```

```
22)   {
23)      return EVAL_PAGE;
24)   }
25)   public void release(){}
26)   public Tag getParent()
27)   {
28)      return null;
29)   }
30) }
```

```
cust1
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

Output:

```
Hello this is Demo JSP
Hello this is from tag handler
This is body of the custom tag
This is after the custom tag invocation
```

If doStartTag() method resturns SKIP_BODY and doEndTag() method returns SKIP_PAGE then the output is:

```
Hello this is Demo JSP
Hello this is from tag handler
```

How to map taglib directive with TLD file

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

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

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

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

```
1)  <web-app>
2)    <jsp-config>
3)        <taglib>
4)        <taglib-uri>www.durgasoft.com</taglib-uri>
5)        <taglib-location>/WEB-INF/MyTld.tld</taglib-location>
6)      </taglib>
7)    </jsp-config>
```

```
8)  </web-app>
```

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

```
1)  <%@ taglib  prefix="mine" uri="www.durgajobs.com" %>
2)
3)  <taglib version="2.1" >
4)    <tlib-version>1.2</tlib-version>
5)      <uri>www.durgajobs.com</uri>
6)      <tag>
7)       <name>mytag</name>
8)       <tag-class>tags.MyCustomTag</tag-class>
9)      </tag>
10) </taglib>
```

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

3 ways

Structure of TLD file:

```
1)  <taglib version="2.1" >
2)    <tlib-version>1.2</tlib-version>
3)
4)      <uri>www.durgajobs.com</uri>
5)
6)      <tag>
7)       <description>This custom tag for weather report</description>
8)       <name>mytag</name>
9)       <tag-class>tags.MyCustomTag</tag-class>
10)      <body-content>xxx</body-content>
11)      <attribute>
12)          <name>
13)        <required>
14)        <rtexprevalue>
15)      </attribute>
16)    </tag>
17)
18) </taglib>
```

<body-content>:

It describes the type of content allowed in tag body.
The allowed values are:

1.empty:

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

```
<mine:mytag>
</mine:mytag>
```

OR

```
<mine:mytag/>
```

2.tagdependent:

Total tag body will be treated as plain text.

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

3.scriptlet:

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

4.jsp:

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

Attributes:

A tag can contain any number of attributes. we can declare these attributes by using <attribute> tag in tld.

<attribute> tag contains the following child tags.

1.<name> - Name of the attribute

2.<required>
- true ➔ means attribute is mandatory
- false ➔ means attribute is optional
- default value is false

3.<rtexprvalue>
runtime expression value
true ➔ means runtime expressions are allowed

Eg: <mine:mytag color="${param.color}" />
false➔ Runtime expressions are not allowed and we have to provide only literals.

Eg: <mine:mytag color="red" />
default value is false.

Things to remember about tag attributes:

A tag can contain attributes also. For each attribute we have to do the following things

1.We have to declare that attribute in tld by using <attribute> tag.

2.For each attribute in TagHandler class, we have to define one instance variable and corresponding setter method.

3.In the case of optional attributes there may be a chance of NullPointerException. We have to handle carefully.

Demo Program for empty Custom Tag with mandatory attribute:

test.jsp:

```
1)  <%@taglib  prefix="mine" uri="/WEB-INF/MyTld.tld" %>
2)  <mine:mytag number="5"/>
3)  <mine:mytag number="${param.num}"/>
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)    <tlib-version>1.2</tlib-version>
3)      <tag>
4)       <name>mytag</name>
5)       <tag-class>tags.MyCustomTag</tag-class>
6)       <body-content>empty</body-content>
7)       <attribute>
8)          <name>number</name>
9)        <required>true</required>
10)      <rtexprvalue>true</rtexprvalue>
11)      </attribute>
12)    </tag>
13)  </taglib>
```

MyCustomTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  public class MyCustomTag implements Tag
5)  {
6)     private int number;
7)     private PageContext pageContext;
8)     public void setPageContext(PageContext pageContext)
9)     {
10)       this.pageContext = pageContext;
```

```
}
public void setParent(Tag t)
{
}
public void setNumber(int number)
{
   this.number=number;
}
public int doStartTag() throws JspException
{
   try{
   JspWriter out = pageContext.getOut();
   out.println("<h1>Double of "+number+" is :"+ (2*number)+"</h1>");
   }
   catch(java.io.IOException e){}
   return SKIP_BODY;
}
public int doEndTag() throws JspException
{
   return EVAL_PAGE;
}
public void release()
{
}
public Tag getParent()
{
   return null;
}
}
```

```
cust2
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

Demo Program for empty Custom Tag with optional attribute:

test.jsp:

```
<%@taglib  prefix="mine1" uri="/WEB-INF/MyTld.tld" %>
<mine1:greeting name="Durga" />
<mine1:greeting/>
```

MyTld.tld:

```
<taglib version="2.1" >
```

```
2)    <tlib-version>1.2</tlib-version>
3)      <tag>
4)       <name>greeting</name>
5)       <tag-class>tags.MyCustomTag</tag-class>
6)       <body-content>empty</body-content>
7)       <attribute>
8)         <name>name</name>
9)        <required>false</required>
10)     </attribute>
11)    </tag>
12) </taglib>
```

MyCustomTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  public class MyCustomTag implements Tag
5)  {
6)     private PageContext pageContext;
7)     private String name;
8)     public void setPageContext(PageContext pageContext)
9)     {
10)       this.pageContext = pageContext;
11)    }
12)    public void setParent(Tag t)
13)    {
14)    }
15)    public void setName(String name)
16)    {
17)       this.name=name;
18)    }
19)    public int doStartTag() throws JspException
20)    {
21)       try
22)       {
23)       JspWriter out = pcontext.getOut();
24)       if(name == null)
25)       {
26)          out.println("<h1>Hello Guest..Good Morning...</h1>");
27)       }
28)       else
29)       {
30)          out.println("<h1>Good Morning..."+name+"</h1>");
31)       }
32)       }
33)       catch(java.io.IOException e){}
34)       return SKIP_BODY;
35)    }
```

```
36)    public int doEndTag() throws JspException
37)    {
38)       return EVAL_PAGE;
39)    }
40)    public void release()
41)    {
42)    }
43)    public Tag getParent()
44)    {
45)       return null;
46)    }
47) }
```

```
cust3
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

IterationTag(I)

IterationTag is the child interface of Tag.

If we want to include tag body multiple times then we should go for IterationTag.

It contains only one extra method doAfterBody() and one extra constant EVAL_BODY_AGAIN.

doAfterBody():

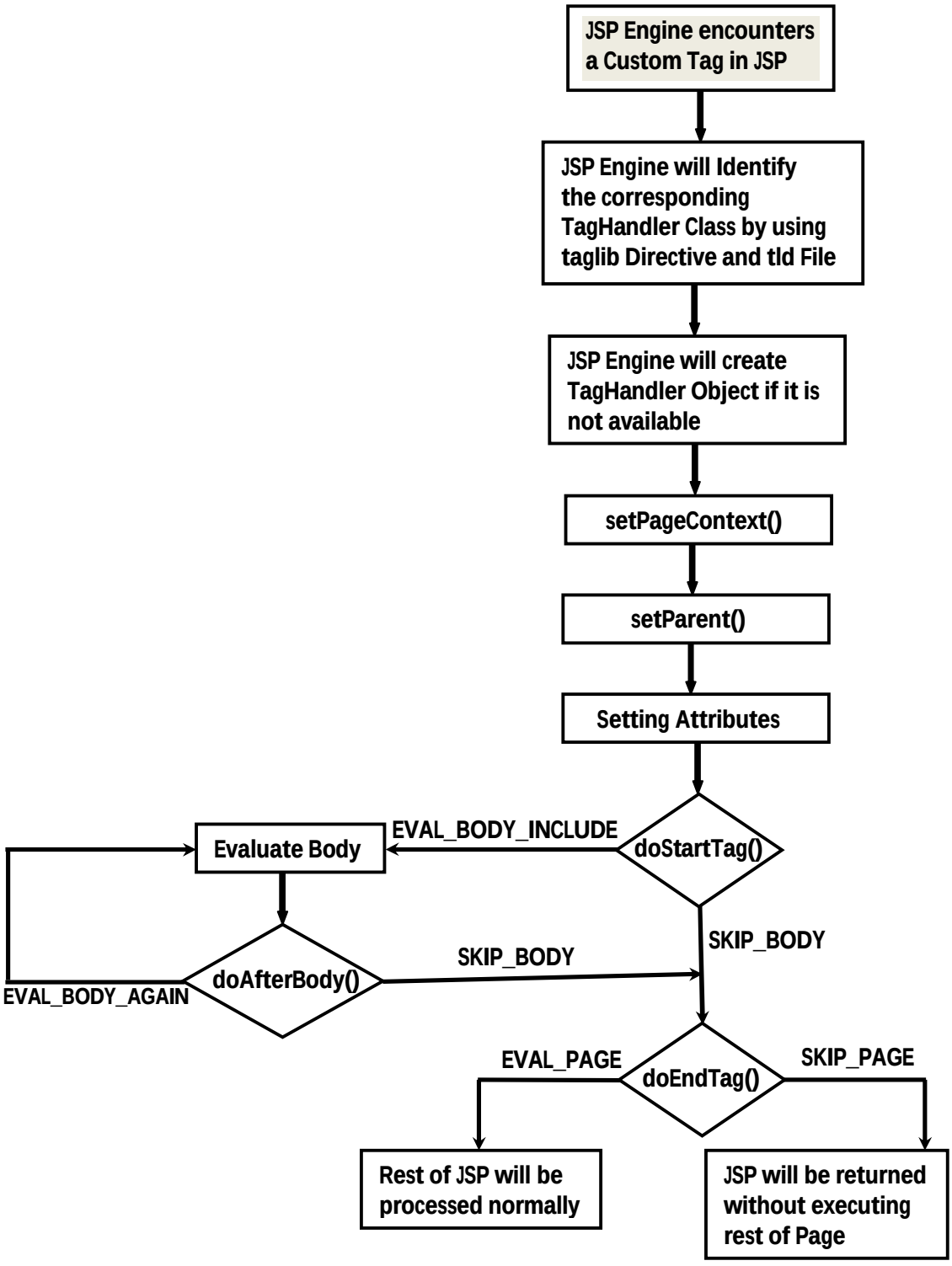
```
public int doAfterBody() throws JspException
```

This method will be executed after doStartTag() method and can return either EVAL_BODY_AGAIN or SKIP_BODY.

If it returns EVAL_BODY_AGAIN then the tag body will be considered once again followed by execution of doAfterBody().

If it returns SKIP_BODY then the body will be skipped and jsp engine calls doEndTag() method.

Flow Chart for life cycle of IterationTag



Demo Program for IterationTag:

test.jsp:

```
1)  <%@taglib  prefix="mine1" uri="/WEB-INF/MyTld.tld" %>
2)  <mine1:loop count="6">
3)   <h1>Learning custom tags is very easy...</h1>
4)  </mine1:loop>
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)   <tlib-version>1.2</tlib-version>
3)      <tag>
4)         <name>loop</name>
5)         <tag-class>tags.MyCustomTag</tag-class>
6)         <attribute>
7)        <name>count</name>
8)        <required>true</required>
9)        </attribute>
10)    </tag>
11)  </taglib>
```

MyCustomTag.java:

```
1)   package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  public class MyCustomTag implements IterationTag
5)  {
6)     private int count;
7)     public void setCount(int count)
8)     {
9)        this.count = count;
10)    }
11)    public void setPageContext(PageContext p)
12)    {
13)    }
14)    public void setParent(Tag t)
15)    {
16)    }
17)    public int doStartTag() throws JspException
18)    {
19)       if(count >0)
20)          return EVAL_BODY_INCLUDE;
21)       else return SKIP_BODY;
22)    }
23)    public int doAfterBody() throws JspException
24)    {
```

```
25)       if(--count >0)
26)          return EVAL_BODY_AGAIN;
27)       else return SKIP_BODY;
28)    }
29)
30)    public int doEndTag() throws JspException
31)    {
32)       return EVAL_PAGE;
33)    }
34)    public void release()
35)    {
36)    }
37)    public Tag getParent()
38)    {
39)       return null;
40)    }
41) }
```

```
cust4
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

TagSupport Class

We can develop Tag Handler class by implementing Tag and IterationTag interfaces directly. But the problem in this approach is we have to provide implementation for all methods even though most of the times our requirement is only doStartTag(), doAfterBody() and doEndTag() methods.

To overcome this problem we should go for TagSupport class. TagSupport class implements IterationTag interface and provides default implementation for all its methods.

The main advantage of extending TagSupport class is we have to override only required methods instead implementing all methods.

Internal implementation of TagSupport class:

```
1)  public class TagSupport
2)     implements IterationTag, Serializable
3)  {
4)
5)     private Tag parent;
6)     protected transient PageContext pageContext;
7)
```

```
8)    public void setPageContext(PageContext pageContext)
9)    {
10)      this.pageContext = pageContext;
11)   }
12)   public void setParent(Tag t)
13)   {
14)      parent = t;
15)   }
16)   public int doStartTag()throws JspException
17)   {
18)      return SKIP_BODY;
19)   }
20)   public int doAfterBody() throws JspException
21)   {
22)      return SKIP_BODY;
23)   }
24)   public int doEndTag() throws JspException
25)   {
26)      return EVAL_PAGE;
27)   }
28)   public Tag getParent()
29)   {
30)      return parent;
31)   }
32)   public void release()
33)   {
34)    pageContext=null;
35)    parent=null;
36)   }
37)   ..
38) }
```

Note:

1. The default return type of doStartTag() and doAfterBody() methods is SKIP_BODY.

2.The default return type of doEndTag() is EVAL_PAGE

3. pageContext variable is by default available to the child class. Hence we can use this variable directly in our Tag Handler class.

Demo Program-1 by using TagSupport class:

test.jsp:

```
1)  <%@taglib  prefix="mine" uri="/WEB-INF/MyTld.tld" %>
2)  <mine:loop count="7">
3)     <h1>Learning custom tags is very easy...</h1>
4)  </mine:loop>
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)   <tlib-version>1.2.3.4</tlib-version>
3)     <tag>
4)      <name>loop</name>
5)      <tag-class>tags.MyCustomTag</tag-class>
6)      <attribute>
7)        <name>count</name>
8)     <required>true</required>
9)     </attribute>
10)   </tag>
11) </taglib>
```

MyCustomTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  public class MyCustomTag extends TagSupport
5)  {
6)     private int count;
7)     public void setCount(int count)
8)     {
9)        this.count = count;
10)    }
11)    public int doStartTag() throws JspException
12)    {
13)       if(count >0)
14)          return EVAL_BODY_INCLUDE;
15)       else return SKIP_BODY;
16)    }
17)    public int doAfterBody() throws JspException
18)    {
19)       if(--count >0)
20)          return EVAL_BODY_AGAIN;
21)       else return SKIP_BODY;
22)    }
23) }
```

```
cust5
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

Demo Program-2 by using TagSupport class:

test.jsp:

```
1)  <%@taglib  prefix="mine" uri="/WEB-INF/MyTld.tld" %>
2)  <mine:mytag/>
3)  <mine:mytag/>
4)  <mine:mytag/>
5)  <mine:mytag/>
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)    <tlib-version>1.2</tlib-version>
3)      <tag>
4)        <name>mytag</name>
5)        <tag-class>tags.MyCustomTag</tag-class>
6)      </tag>
7)    </taglib>
```

MyCustomTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  import java.io.*;
5)  public class MyCustomTag extends TagSupport
6)  {
7)     public int doStartTag() throws JspException
8)     {
9)        try{
10)       JspWriter out = pageContext.getOut();
11)       out.println("<h1>Hello this is from tag handler</h1>");
12)       }catch(IOException e){}
13)       return EVAL_BODY_INCLUDE;
14)    }
15) }
```

```
cust5A
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

BodyTag(I)

It is the child interface of IterationTag.

If we want to manipulate Tag Body then we should go for BodyTag interface.

BodyTag interface defines the following 2 extra methods.

1. public void setBodyContent(BodyContent b)
2. public void doInitBody() throws JspException

BodyTag interface defines the following 2 extra constants

1.EVAL_BODY_BUFFERED
2.EVAL_BODY_TAG==>Deprecated in JSP 1.2V

BodyContent:

It is an abstract class and extends JspWriter.

BodyContent object represents body of the Tag.

BodyContent class defines the following methods to retrieve tag body.

1. public String getString()
   returns body content as String

2. public Reader getReader()
   returns Reader object to read tag body

3. public JspWriter getEnclosingWriter()
   Returns JspWriter object of Parent class
   If there is no parent class then it returns current jsp out object.

4. public void clearBody()
   It clears bodyContent object. i.e data present in bodyContent object will be removed.

Life cycle of Tag Handler that implements BodyTag:

The life cycle of BodyTag Handler is exactly similar to IterationTag Hanlder. The difference is doStartTag() method can return EVAL_BODY_BUFFERED in addition to EVAL_BODY_INCLUDE and SKIP_BODY.

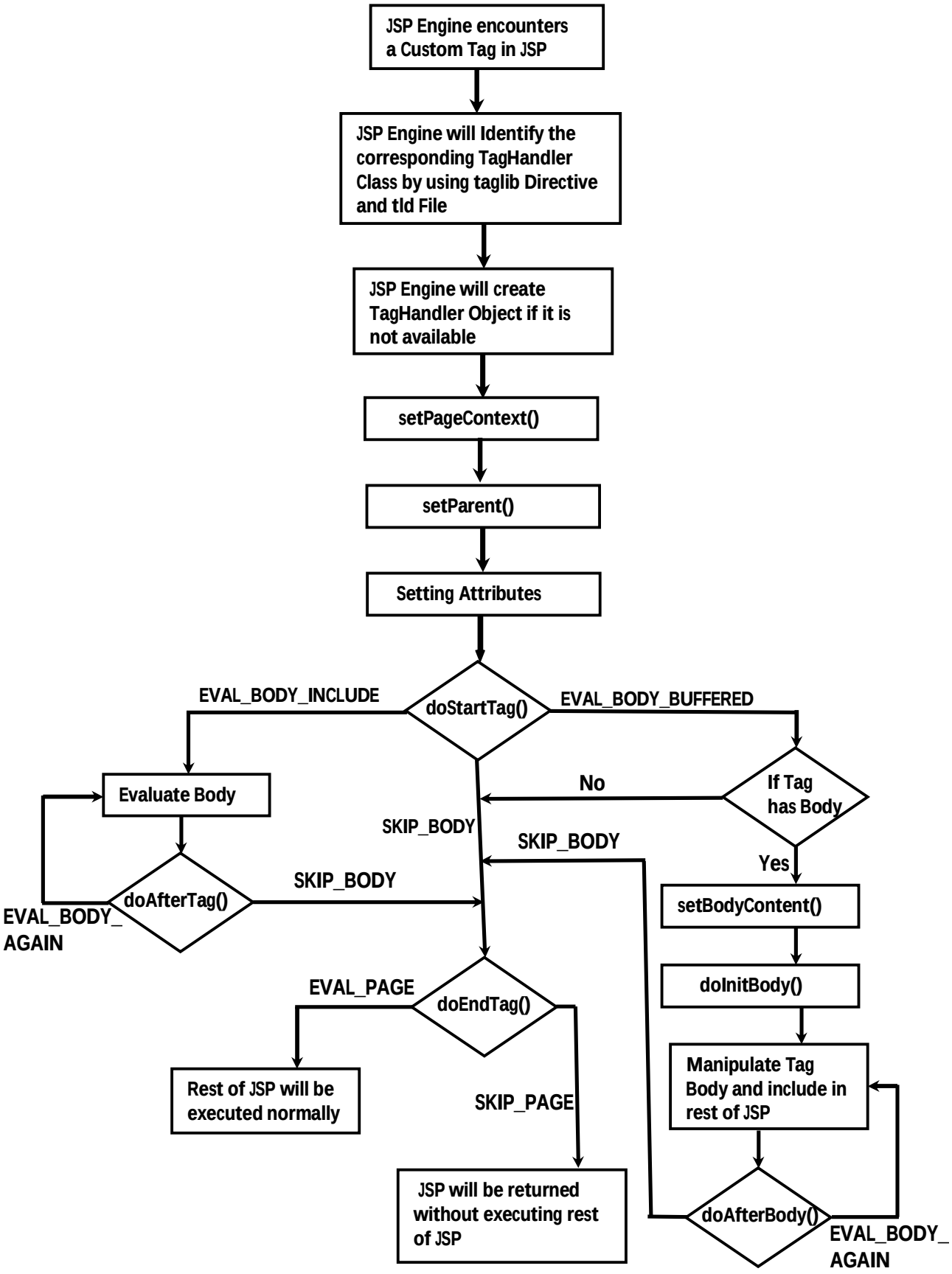
If the method returns EVAL_BODY_INCLUDE then body will be evaluated and included exactly similar to IterationTag.

If the method returns EVAL_BODY_BUFFERED and tag contains body then JSP engine creates BodyContent object and call setBodyContent() method followed by doInitBody() method.

Note:

setBodyContent() and doInitBody() method won't be executed in the following case:
If doStartTag() returns either EVAL_BODY_INCLUDE or SKIP_BODY or if the tag does not contain body.

Flow Chart for BodyTag Life Cycle



BodyTagSupport Class

This class extends TagSupport class and implements BodyTag interface.

This class provides default implementation for all 9 methods available in BodyTag interface.

It is very easy to develop tag handler by extending BodyTagSupport class instead of implementing BodyTag interface directly. In this case we have to provide implementation only for required methods instead of implementing all 9 methods.

Internal implementation of BodyTagSupport class:

```
1)  public class BodyTagSupport extends TagSupport implements BodyTag
2)  {
3)     protected transient BodyContent bodyContent;
4)     protected transient PageContext pageContext;
5)     private Tag parent;
6)     public void setPageContext(PageContext pageContext)
7)     {
8)        this.pageContext = pageContext;
9)     }
10)    public void setParent(Tag t)
11)    {
12)       parent = t;
13)    }
14)    public int doStartTag() throws JspException
15)    {
16)       return EVAL_BODY_BUFFERED;
17)    }
18)    public void setBodyContent(BodyContent b)
19)    {
20)       bodyContent = b;
21)    }
22)    public void doInitBody() throws JspException
23)    {
24)    }
25)    public int doAfterBody() throws JspException
26)    {
27)       return SKIP_BODY;
28)    }
29)    public int doEndTag() throws JspException
30)    {
31)       return EVAL_PAGE;
32)    }
33)    public BodyContent getBodyContent()
34)    {
35)       return bodyContent;
36)    }
```

```
37)    ....
38) }
```

Note:

1.pageContext and bodyContent variables are by default available to our Tag handler class and hence we can use these directly.

2. The default return type of
doStartTag() is EVAL_BODY_BUFFERED,
doAfterBody() is SKIP_BODY ,
doEndTag() is EVAL_PAGE.

Demo Program for BodyTag and BodyTagSupport:

test.jsp:

```
1)  <%@taglib  prefix="mine1" uri="/WEB-INF/MyTld.tld" %>
2)  <mine1:body >
3)   <h1>Learning custom tags is very easy...</h1>
4)  </mine1:body>
5)  <mine1:body >
6)   <h1>Durga Software Solutions....</h1>
7)  </mine1:body>
8)  <mine1:body >
9)   <h1>Ameerpet, Hyderabad</h1>
10) </mine1:body>
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)   <tlib-version>1.2</tlib-version>
3)     <tag>
4)      <name>body</name>
5)      <tag-class>tags.MyCustomTag</tag-class>
6)     </tag>
7)  </taglib>
```

MyCustomTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  public class MyCustomTag extends BodyTagSupport
5)  {
6)     public int doAfterBody() throws JspException
7)     {
8)        try
```

```
9)        {
10)          String s = bodyContent.getString();
11)          s = s.toLowerCase();
12)          JspWriter out = bodyContent.getEnclosingWriter();
13)          out.println(s);
14)       }
15)       catch(Exception e){}
16)       return SKIP_BODY;
17)    }
18) }
```

```
cust6
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

Comparison b/w TagSupport and BodyTagSupport Classes

Method	Tag Support	BodyTagSupport
1) doStartTag() 1. Possible Return Values	EVAL_BODY_INCLUDE, SKIP_BODY	EVAL_BODY_INCLUDE, SKIP_BODY, EVAL_BODY_BUFFERED
2. Default Value from Implementation Class	SKIP_BODY	EVAL_BODY_BUFFERED
3. Number of times it can be called per Tag	Only Once	Only Once
2) doAfterBody() 1. Possible Return Values	EVAL_BODY_AGAIN, SKIP_BODY	EVAL_BODY_AGAIN, SKIP_BODY
2. Default Value from Implementation Class	SKIP_BODY	SKIP_BODY
3. Number of times it can be called per Tag	Zero OR More	Zero OR More
3) doEndTag() 1. Possible Return Values	EVAL_PAGE, SKIP_PAGE	EVAL_PAGE, SKIP_PAGE
2. Default Value from Implementation Class	EVAL_PAGE	EVAL_PAGE
3. Number of times it can be called per Tag	Only Once	Only Once
4) setBodyContent() doInitBody() Circumstances under which these Methods can be called and Number of times per Tag Invocation	Not Applicable	Executed only once iff doStartTag() returns EVAL_BODY_BUFFERED and Tag contains Body

Co-operative OR Nested tags:

Sometimes a group of tags work together to perform required functionality. Such type of tags are called co-operative or nested tags.

Eg:
In JSTL <c:choose>,<c:when> and <c:otherwise> tags work together to implement java's switch statement.
These tags are called co-operative tags.

Eg:

```
<mine:outerTag>
   <mine:innerTag/>
</mine:outerTag>
```

From Inner Tag(Child Tag) we can get Parent Tag Referece by using getParent() method.

Tag parent=getParent();

Demo Program1 for Nested Tags:

test.jsp:

```
1)  <%@taglib  prefix="mine" uri="/WEB-INF/MyTld.tld" %>
2)     <mine:mytag>
3)        <mine:mytag>
4)           <mine:mytag>
5)              <mine:mytag/>
6)           </mine:mytag>
7)        </mine:mytag>
8)     </mine:mytag>
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)   <tlib-version>1.2</tlib-version>
3)     <tag>
4)      <name>mytag</name>
5)      <tag-class>tags.MyCustomTag</tag-class>
6)      <body-content>JSP</body-content>
7)    </tag>
8)  </taglib>
```

MyCustomTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  public class MyCustomTag extends TagSupport
5)  {
6)     public int doStartTag() throws JspException
7)     {
8)        int level = 0;
9)        Tag t = getParent();
10)       while( t != null)
11)       {
12)          level++;
13)          t = t.getParent();
14)       }
```

```
15)       try{
16)          JspWriter out = pageContext.getOut();
17)          out.println("<h1>Nested level is :"+level+"</h1>");
18)       }
19)       catch(java.io.IOException e){}
20)       return EVAL_BODY_INCLUDE;
21)    }
22) }
```

```
cust7
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

Demo Program2 for Nested Tags:

test.jsp:

```
1)  <%@taglib  prefix="mine" uri="/WEB-INF/MyTld.tld" %>
2)  <mine:menu>
3)    <mine:menuitem item="chicken65"/>
4)    <mine:menuitem item="Mutton"/>
5)    <mine:menuitem item="Fish"/>
6)  </mine:menu >
7)
8)  <mine:menu>
9)    <mine:menuitem item="CoreJava"/>
10)   <mine:menuitem item="AdvJava"/>
11)   <mine:menuitem item="Oracle"/>
12)   <mine:menuitem item="Spring"/>
13)   <mine:menuitem item="Hibernate"/>
14)   <mine:menuitem item="WebServices"/>
15)   <mine:menuitem item="DP"/>
16)   <mine:menuitem item="RT"/>
17) </mine:menu >
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)   <tlib-version>1.2</tlib-version>
3)
4)    <tag>
5)      <name>menu</name>
6)      <tag-class>tags.MenuTag</tag-class>
7)      <body-content>JSP</body-content>
8)    </tag>
```

```
9)
10)    <tag>
11)      <name>menuitem</name>
12)      <tag-class>tags.MenuItemTag</tag-class>
13)      <attribute>
14)        <name>item</name>
15)     <required>true</required>
16)      </attribute>
17)    </tag>
18)
19)  </taglib>
```

MenuTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  import java.util.*;
5)  public class MenuTag extends TagSupport
6)  {
7)     private ArrayList l = null;
8)     public int doStartTag() throws JspException
9)     {
10)       l = new ArrayList();
11)       return EVAL_BODY_INCLUDE;
12)    }
13)    public void addMenuItem(String s)
14)    {
15)       l.add(s);
16)    }
17)    public int doEndTag() throws JspException
18)    {
19)       try
20)       {
21)       JspWriter out = pageContext.getOut();
22)       out.println("<h1><br>Menu Items are  :" +l+"</h1>");
23)       }
24)       catch(Exception e ) {}
25)
26)          return EVAL_PAGE;
27)    }
28) }
```

MenuItemTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  import java.util.*;
```

```
5)  public class MenuItemTag extends TagSupport
6)  {
7)     private String item;
8)     public void setItem(String item)
9)     {
10)       this.item = item;
11)    }
12)    public int doStartTag() throws JspException
13)    {
14)       MenuTag parent = (MenuTag)getParent();
15)       parent.addMenuItem(item);
16)       return SKIP_BODY;
17)    }
18) }
```

```
cust8
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
      |-tags
            |-MenuTag.class
            |-MenuItemTag.class
```

Getting an arbitrary Ancestor class:

We can get immediate parent by using getParent() method.

TagSupport class contains the following method to get an arbitrary ancestor class.

```
public static Tag findAncestorWithClass(Tag t,Class c)
```

Accessing JSP Implicit Objects in Tag Handler Class:

By using pageContext implicit object we can access all other implicit objects in Tag handler class.

Tag Handler class can get pageContext object as argument to setPageContext() method.

PageContext class defines the following methods to get all other JSP implicit objects.

1) request → getRequest()
2) response → getResponse()
3) config → getServletConfig()
4) application → getServletContext()
5) session() → getSession()
6) out → getOut()

7) page → getPage()
8) exception → getException()

Note:

Exception implicit object is available only in error pages. If the enclosing JSP is not error page then getException() returns null

Demo Program for JSP Implicit objects:

test.jsp:

```
1) <%@taglib  prefix="mine1" uri="/WEB-INF/MyTld.tld" %>
2) <%@ page isErrorPage="true" %>
3) <mine1:myTag/>
```

test1.jsp:

```
1) <%@ page errorPage="test.jsp" %>
2) <% System.out.println(10/0); %>
```

MyTld.tld:

```
1) <taglib version="2.1" >
2)  <tlib-version>1.2</tlib-version>
3)
4)    <tag>
5)      <name>myTag</name>
6)      <tag-class>tags.MyCustomTag</tag-class>
7)      <body-content>JSP</body-content>
8)    </tag>
9)  </taglib>
```

MyCustomTag.java:

```
1) package tags;
2) import javax.servlet.jsp.*;
3) import javax.servlet.jsp.tagext.*;
4) import javax.servlet.*;
5) import javax.servlet.http.*;
6) public class MyCustomTag extends TagSupport
7) {
8)    public int doStartTag() throws JspException
9)    {
10)      ServletRequest req = pageContext.getRequest();
11)      String s1 = req.getServerName()+":"+req.getServerPort();
12)      ServletResponse resp = pageContext.getResponse();
13)      String s2 = resp.getContentType();
```

```
14)       HttpSession session = pageContext.getSession();
15)       String s3 = session.getId();
16)       Throwable e = pageContext.getException();
17)       try{
18)       JspWriter out = pageContext.getOut();
19)       out.println("<h1>"+s1);
20)       out.println(s2);
21)       out.println(s3);
22)       out.println(e+"</h1>");
23)       }
24)       catch(Exception e1){}
25)       return EVAL_BODY_INCLUDE;
26)    }
27) }
```

```
cust9
  |-test.jsp
  |-test1.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

http://localhost:7777/cust9/test1.jsp
http://localhost:7777/cust9/test.jsp

Getting and Setting Attributes by using PageContext API

PageContext class defines the following methods to perform attribute management.

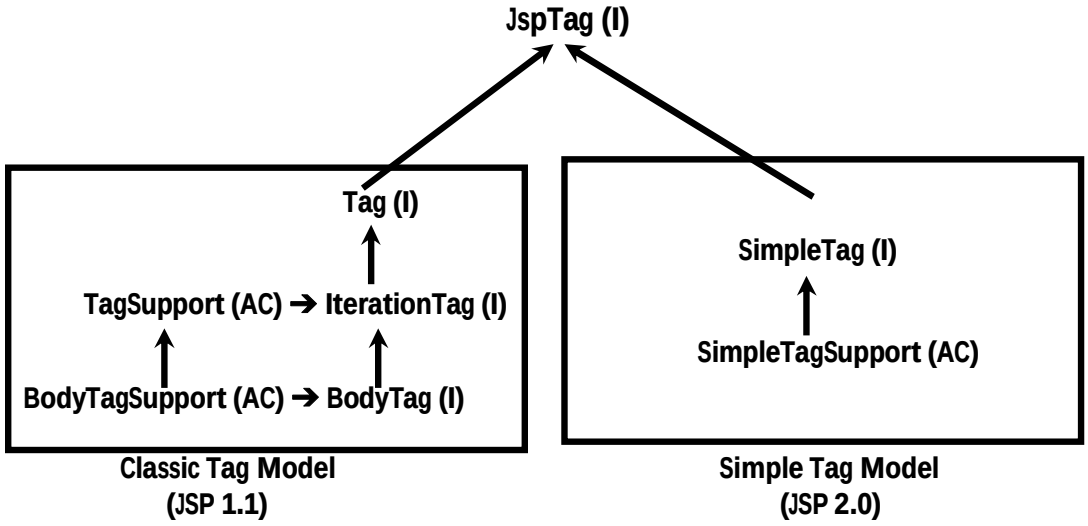
1. public void setAttribute(String name,Object value)
2. public void setAttribute(String name,Object value,int scope)
3. public Object getAttribute(String name)
4. public Object getAttribute(String name,int scope)
5. public void removeAttribute(String name)
6. public void removeAttribute(String name,int scope)
7. public Object findAttribute(String name)
8. public Enumeration getAttributeNamesInScope(int scope)

SimpleTag Model

Implementing custom tags by using classic tag model (Tag, IterationTag, BodyTag, TagSupport, BodyTagSupport)is very complex because each tag has its own life cycle and different possible return types for each method.

To resolve this complexity, Sun people introduced Simple Tag Model in JSP 2.0V

In this model, we can build custom tags by using SimpleTag interface and its implementation class SimpleTagSupport.



SimpleTag(I):

It is the child interface of JspTag and contains the following 5 methods...

1. public void setJspContext(JspContext c)
2. public void setParent(JspTag t)
3. public void setJspBody(JspFragment f)
4. public void doTag()throws JspException,IOException
5. public JspTag getParent()

Life Cycle of SimpleTag Handler:

1.whenever JSP engine encounters a custom tag in JSP,it will identify the corresponding Tag Hanlder class by using taglib directive and tld file.JSP Engine creates a new instance of Tag Handler class by executing public no-arg constructor. Simple Tag objects are never reused by web container. Hence for each tag invocation a new Tag Handler object will be created.

2. Web container executes setJspContext() method to make JspContext object available to Tag Handler class.By using this JspContext object Tag handler class can get all JSP implicit objects and attributes.

3. Web container executes setParent() to make Parent Tag object available to Tag Handler class. This method is useful in the case of nested tags.

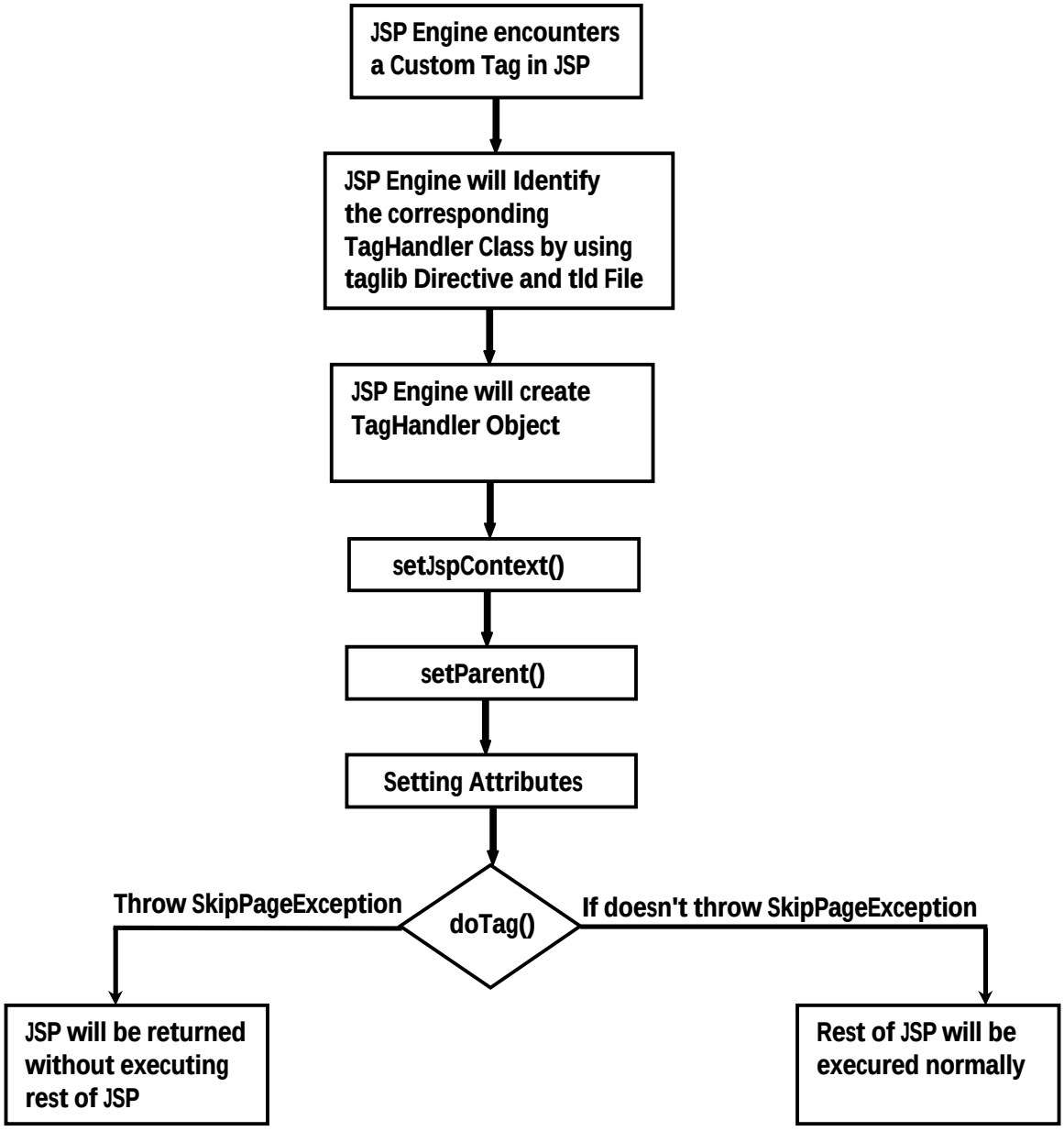
4.If custom tag invoked with attributes then jsp engine will call corresponding setter methods to set attribute values.

5.If the tag is invoked with body then setJspBody() method will be executed by taking JspFragment object as argument. JspFragment represents tag body in Simple Tag Model.
In Simple Tag Model Tag body should not contain any scripting elements.

6. Finally Jsp engine executes doTag() method which is responsible to provide required functionality. This method is equivalent to doStartTag(),doEndTag() and doAfterBody() methods in classic tag model. EVAL_BODY_INCLUDE

7. Once doTag() method completes ,the Tag Handler object will be destroyed by web container.

Flow chart for Simple Tag Handler Life Cycle



SimpleTagSupport class

It is the implementation class of SimpleTag interface.

It provides default implementation for all methods of SimpleTag interface.

This class also provides the following extra methods

1. public JspContext getJspContext()
2. public JspFragment getJspBody()
3. public JspTag findAncestorWithClass(JspTag t,Class c)

Demo Program for SimpleTagSupport:

test.jsp:

```
1)  <%@taglib  prefix="mine" uri="/WEB-INF/MyTld.tld" %>
2)  <h1>This is Simple Tag Demo<br>
3)  <mine:mytag/>
4)  <h1>This is rest of the JSP</h1>
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)   <tlib-version>1.2</tlib-version>
3)     <tag>
4)       <name>mytag</name>
5)       <tag-class>tags.MyCustomTag</tag-class>
6)       <body-content>tagdependent</body-content>
7)     </tag>
8)  </taglib>
```

MyCustomTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  import java.io.*;
5)  public class MyCustomTag extends  SimpleTagSupport
6)  {
7)     public void doTag() throws JspException,IOException
8)     {
9)        JspWriter out = getJspContext().getOut();
10)       out.println("<h1>Hello this is from simple tag handler</h1>");
11)       //throw new SkipPageException();
12)    }
13) }
```

```
cust10
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
      |-tags
            |-MyCustomTag.class
```

If doTag() method throws SkipPageException then the output is:

```
This is Simple Tag Demo
Hello this is from simple tag handler
```

If doTag() method does not throw SkipPageException then the output is:

```
This is Simple Tag Demo
Hello this is from simple tag handler
This is rest of the JSP
```

Q. What is the difference b/w classic and simple tags wrt tag body?

In classic tag model we are allowed to take scripting elements in tag body. Hence the possible values for <body-content> tag are : empty, tagdependent, scriptless, jsp.
default value is JSP.

But in Simple Tag Model Scripting elements are not allowed in tag body. Hence allowed values for <body-content> are empty, tagdepdendent, scriptless but not jsp.

Default value is scriptless

Processing Body Content in Simple Tags:

We can access Tag Body inside Simple Tag Handler by using getJspBody() method. This method returns JspFragment object.

```
public JspFragment getJspBody()
```

JspFragment object represents Tag Body. On this object we can call the following 2 methods.

1. public JspContext getJspContext()

2. public void invoke(Writer w);

   It causes evaluation of TagBody and return to supplied writer.
   use "null" argument to write directly to the JspOutputStream.

Demo Program for SimpleTag to process Tag Body:

test.jsp:

```
1)  <%@taglib  prefix="mine" uri="/WEB-INF/MyTld.tld" %>
2)  <h1> This is Before tag invocation<br>
3)
4)  <mine:mytag>
5)     <h1>This is Tag Body<br>
6)  </mine:mytag>
7)
8)  <h1>This is After tag invocation</h1>
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)   <tlib-version>1.2</tlib-version>
3)     <tag>
4)      <name>mytag</name>
5)      <tag-class>tags.MyCustomTag</tag-class>
6)      <body-content>tagdependent</body-content>
7)     </tag>
8)   </taglib>
```

MyCustomTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  import java.io.*;
5)  public class MyCustomTag extends  SimpleTagSupport
6)  {
7)     public void doTag() throws JspException,IOException
8)     {
9)        JspWriter out = getJspContext().getOut();
10)       out.println("<h1>Hello this is from simple tag handler</h1>");
11)       getJspBody().invoke(null);
12)       //throw new SkipPageException();
13)    }
14) }
```

```
cust12
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

Key Differences b/w Simple and Classic Tags

Property	Simple Tags	Classic Tags
1) Tag Interfaces	Simple Tag	Tag ↑ Iteration Tag ↑ Body Tag
2) Supporting Imlementation Classes	SimpleTagSupport	TagSupport ↑ BodyTagSupport
3) Key Life Cycle Methods that we have to implement	doTag()	doStartTag() doEndTag() doAfterBody()
4) How to write Response to JspOutputStream	getJspContext().getOut().println(); No need to use try - catch for IoException	pageContext.getOut().println(); Compulsory we should enclose by using try - catch for IoException
5) How to access JSP Implicit Objects and Attributes	By using JspContext Object	By using PageContext Object
6) How to cause the Body to be processed	getJspBody().invoke(null)	Returns EVAL_BODY_INCLUDE from doStartTag() OR EVAL_BODY_BUFFERED in Body Tag Interface from doStartTag()
7) How to cause the Current Page Evaluation to stop	throw SkipPageException from doTag()	Return SKIP_PAGE from doEndTag()

Dynamic Attributes

In general tags can contain attributes. We can declare these attributes in tld file by using <attribute> tag. In the tag handler class we have to maintain instance variables and corresponding setter methods. Such type of attributes are called static attributes.

But we can use attributes, even though TLD file does not contain any <attribute> tag declarations. Such type of attributes are called Dynamic Attributes.

Dynamic attributes are applicable for both classic and simple tags and introduced in Jsp 2.0version.

To Support dynamic attributes we have to do the following arrangements:

1.In the tld file we have to use <dynamic-attributes> tag

```
<tag>
    ....
    <dynamic-attributes>true</dynamic-attributes>
</tag>
```

2.The corresponding Tag Handler class should implements DynamicAttributes interface.

This interface introduced in JSP 2.0V and contains only one method setDynamicAttribute().

```
public void setDynamicAttribute(String namespace, String name, Object value)
```

web container will call this method for each dynamic attribute.

Demo Program-1 for Dynamic Attributes:

test.jsp:

```
1)  <%@taglib  prefix="mine" uri="/WEB-INF/MyTld.tld" %>
2)
3)  <mine:mytag  name="pawan"  age="45" wife1="Nandita" wife2="Renu" wife3="Anna Lezi
    " party="JanaSena" address="hyd" />
```

MyTld.tld:

```
1)  <taglib version="2.1" >
2)   <tlib-version>1.2</tlib-version>
3)     <tag>
4)       <name>mytag</name>
5)       <tag-class>tags.MyCustomTag</tag-class>
6)       <body-content>tagdependent</body-content>
```

```
    <dynamic-attributes>true</dynamic-attributes>
  </tag>
</taglib>
```

MyCustomTag.java:

```
package tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.util.*;
public class MyCustomTag extends  SimpleTagSupport implements DynamicAttributes
{
   HashMap h = new HashMap();
   public void setDynamicAttribute(String ns, String name, Object value)
   {
      h.put(name,value);
   }
   public void doTag() throws JspException,IOException
   {
      JspWriter out = getJspContext().getOut();
      out.println("<h1>"+h+"</h1>");
   }
}
```

```
cust11D
  |-test.jsp
  |-WEB-INF
    |-MyTld.tld
    |-classes
       |-tags
            |-MyCustomTag.class
```

Demo Program-2 for Dynamic Attributes:

test.jsp:

```
<%@taglib  prefix="mine" uri="/WEB-INF/MyTld.tld" %>
<h1>Hello this is Sample JSP</h1>
<mine:mytag num="2" min="10" max="20" pow="2" />
```

MyTld.tld:

```
<taglib version="2.1" >
  <tlib-version>1.2</tlib-version>
    <tag>
      <name>mytag</name>
      <tag-class>tags.MyCustomTag</tag-class>
      <attribute>
```

```
7)          <name>num</name>
8)          <required>true</required>
9)       </attribute>
10)      <dynamic-attributes>true</dynamic-attributes>
11)      <body-content>empty</body-content>
12)   </tag>
13)  </taglib>
```

MyCustomTag.java:

```
1)  package tags;
2)  import javax.servlet.jsp.*;
3)  import javax.servlet.jsp.tagext.*;
4)  import java.io.*;
5)  public class MyCustomTag extends  SimpleTagSupport implements DynamicAttributes
6)  {
7)     double num;
8)     String output = "";
9)     public void setNum(double num)
10)    {
11)       this.num = num;
12)    }
13)    public void setDynamicAttribute(String ns, String name, Object value)
14)    {
15)       double d = Double.parseDouble((String)value);
16)       if(name == "min")
17)       {
18)          output= output + "The Minimum value is :"+Math.min(num,d)+"<br>";
19)       }
20)       else if(name == "max")
21)       {
22)          output= output + "The Maximum value is :"+Math.max(num,d)+"<br>";
23)       }
24)       else if(name == "pow")
25)       {
26)          output= output + "The Power value is :"+Math.pow(num,d)+"<br>";
27)
28)       }
29)    }
30)    public void doTag() throws JspException,IOException
31)    {
32)       JspWriter out = getJspContext().getOut();
33)       out.println("<h1>"+output+"</h1>");
34)    }
35) }
```

cust11
|-test.jsp
|-WEB-INF

```
|-MyTld.tld
|-classes
   |-tags
        |-MyCustomTag.class
```

Tag Files

Objective:

1. Describe the semantics of tag file
2. Describe application structure of Tag Files
3. Write a Tag File and Explain constraints on JspContent in body of tag

Tag Files concept introduced in JSP 2.0V.
Tag File is a simple jsp page or jsp document designed to be used as Custom Tag.

The main advantage of Tag Files is we can build very easily when compared with classic and simple tags.

The main limitation of tag files is it won't suggestible for doing much processing.

Building and using a Simple Tag File:

1. Write a JSP Page or Document and save it with .tag extension.

2.Place this .tag file in /WEB-INF/tags folder.

3. put a taglib directive in the jsp with "tagdir" attribute.

```
<%@ taglib prefix="mine" tagdir="/WEB-INF/tags" %>
```

Demo Program:

test.jsp:

```
1)  <%@taglib  prefix="mine" tagdir="/WEB-INF/tags" %>
2)  <h1>This is Tag File Demo Example</h1>
3)  <mine:mytag/>
4)  <mine:mytag/>
```

mytag.tag:

```
<h1> Hello....This is from the tag file <br>
```

```
tagFileA
  |-test.jsp
  |-WEB-INF
```

```
|-tags
  |-mytag.tag
```

Note: The name of custom tag and tag file should be matched.

Note:

1. Tag Files are internally converted into Simple Tag Handlers and corresponding classes are available in work folder.

2. It is not required to write TLD File.

Declaring a Tag file with attribute:

We can define attributes for tag files by using attribute directive.

```
<%@ attribute name="title" required="true" rtexprvalue="true" %>
```

Demo Program:

test.jsp:

```
1)  <%@taglib  prefix="mine" tagdir="/WEB-INF/tags" %>
2)  <h1> Tag File with Attributes  Example</h1>
3)  <mine:mytag title="kabali"/>
```

mytag.tag:

```
1)  <%@ attribute name="title" required="true" rtexprvalue="true" %>
2)  <h1> Hello....${title} is big flop but business is good 600Cr..</h1>
```

```
tagFileB
  |-test.jsp
  |-WEB-INF
    |-tags
      |-mytag.tag
```

Declaring Body Content for a Tag File:

Tag File invocation can contain body, but we are not allowed to use scripting elements.

We can declare body-content in tag file by using tag directive

```
<%@ tag body-content="tagdependent" %>
```

Allowed values are ➔ empty,tagdependent,scriptless
Default value is : scriptless
In the tag file we can access tag body by using <jsp:doBody/>

Demo Program:

test.jsp:

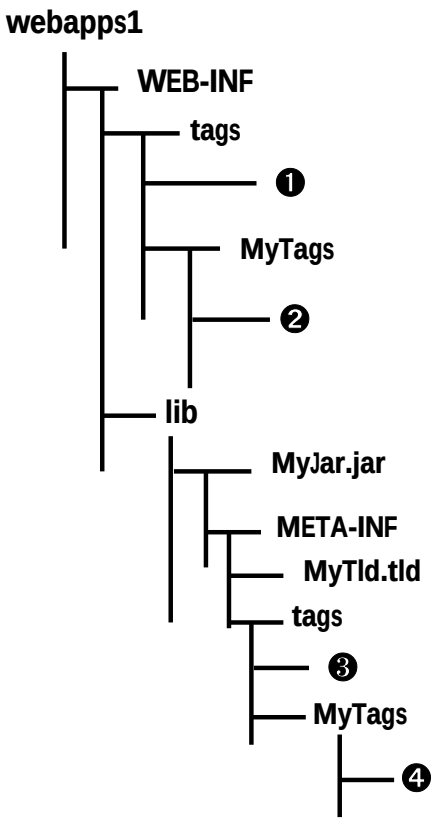
```
1)  <%@taglib  prefix="mine" tagdir="/WEB-INF/tags" %>
2)
3)  <mine:header color="red">
4)   This is Body of tag file
5)  </mine:header>
```

header.tag:

```
1)  <%@ attribute name="color" required="true" rtexprvalue="true" %>
2)  <%@ tag  body-content="tagdependent" %>
3)  <h1> Hello....This is from the tag file<br>
4)
5)  <font color="${color}" > <jsp:doBody/></font> </h1>
```

```
tagFiles1
  |-test.jsp
  |-WEB-INF
    |-tags
      |-mytag.tag
```

Where web container will search for Tag Files:



1. Either directly or indirectly in WEB-INF/tags folder

2. Either directly or indirectly in META-INF/tags folder present in jar file of WEB-INF/lib folder

Note:
Whenever we are deploying tag file in some jar then compulsory we have to write tld file.

Demo Program:

test.jsp:

```
1)  <%@taglib  prefix="mine" uri="www.durgasoft.com" %>
2)  <mine:mytagfile/>
3)  <mine:mytagfile/>
```

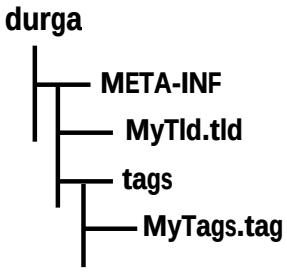
MyTld.tld:

```
1)  <taglib version="2.1" >
2)    <tlib-version>1.2</tlib-version>
3)    <uri>www.durgasoft.com</uri>
4)
5)      <tag-file>
6)        <name>mytagfile</name>
7)        <path>/META-INF/tags/mytag.tag</path>
8)      </tag-file>
9)
10) </taglib>
```

mytag.tag:

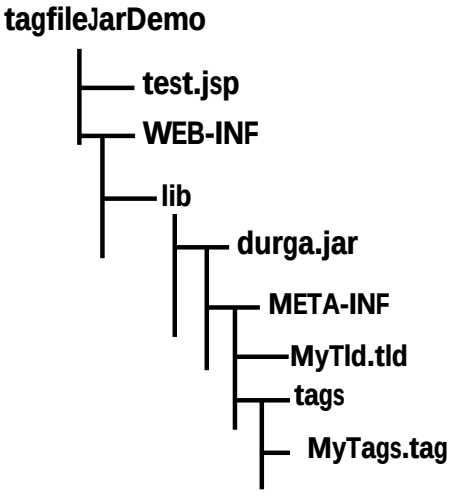
```
<h1> Hello....This is from the tag file deployed in jar file <br>
```

First we have to Create JAR File



1) Create JAR File for durga Folder.
2) jar -cvf durga.jar
3) Place that JAR Folder in lib.

Original Folder Structure



Disabling Scripting Language:

It is not recommended to use scripting elements in JSP. We can disable by using <scripting-invalid> tag in web.xml.

```
1)  <web-app>
2)    <jsp-config>
3)       <jsp-property-group>
4)          <url-pattern>*.jsp</u-p>
5)       <scripting-invalid>true</s-i>
6)       <jsp-property-group>
7)    </jsp-config>
8)  </web-app>
```

Once scripting language is invalidated, we are not allowed to use scripting elements in JSP, otherwise we will get translation time error.(Tomcat won't provide support for this feature)

Similarly we can disable expression language globally.

```
1)  <web-app>
2)    <jsp-config>
3)       <jsp-property-group>
4)          <url-pattern>*.jsp</u-p>
5)       <el-ignored>true</e-i>
6)       </jsp-property-group>
7)    </jsp-config>
8)  </web-app>
```

Top Most Important 3 JSP FAQ's

1) Differences b/w Static Include and Dynamic Include
OR
Differences b/w Include Directive and Include Action

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

2) Explain about JSP Implicit Objects?

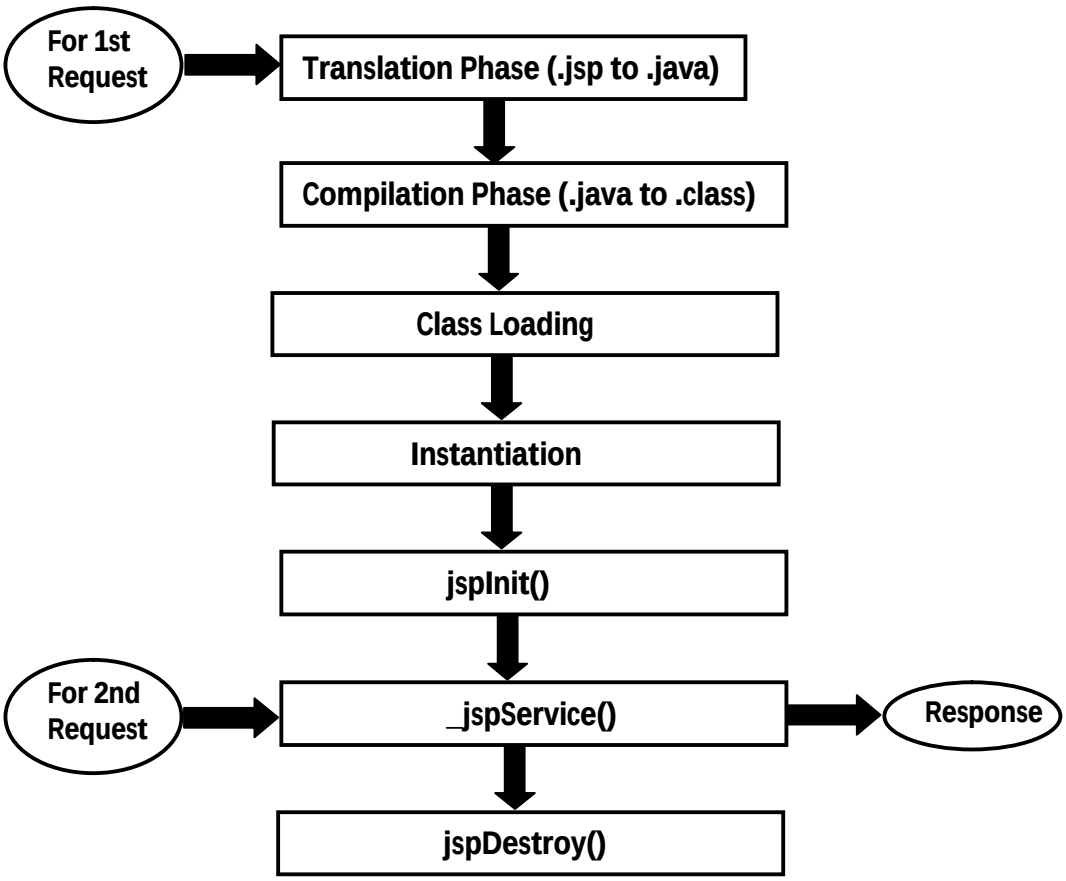
When compared with Servlet Programming, Developing JSPs is very easy because the required mandatory stuff automatically available in every JSP. Implicit objects also one such area ,which are by default available for every JSP.

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

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

3) Explain about Life Cycle of JSP:

- Translation Phase (.jsp➔.java)
- Compilation Phase(.java ➔.class)
- Servlet Class Loading
- Servlet Instantiation
- jspInit()
- _jspService()
- jspDestroy()



JSP FAQ's

1) What is JSP ? Describe its Concept?

Java Server Pages (JSP) is a server side component for the generation of dynamic information as the response. Best suitable to implement view components (presentation layer components). It is part of SUN's J2EE platform.

2) Explain the benefits of JSP?

These are some of the benefits due to the usage of JSP they are:

1) Portability, reusability and logic components of the language can be used across various platforms.
2) Memory and exception management.
3) Has wide range of API which increases the output functionality.
4) Low maintenance and easy deployment.
5) Robust performance on multiple requests.

3) Is JSP technology extensible?

Yes, it is. JSP technology is extensible through the development of custom actions, or tags, which are encapsulated in tag libraries.

4) Can we implement an interface in a JSP?

No

5) What are the advantages of JSP over Servlet?

1) Best suitable for view components
2) We can separate presentation and business logic
3) The JSP author not required to have strong java knowledge
4) If we are performing any changes to the JSP, then not required to recompile and reload explicitly
5) We can reduce development time.

6) Differences between Servlets and JSP?

Servlets	JSP
1) Best suitable for processing Logic	1) Best suitable for presentation Logic
2) We cannot separate Business Logic and Presentation Logic	2) Separation of Presentation Logic and Business Logic is possible
3) Servlet Developer should have Strong Knowledge in Java	3) JSP Author is not required to have Strong Knowledge in Java
4) For Source Code changes ,we have to perform explicitly Compilation	4) For Source Code changes ,it is not required to perform explicit Compilation
5) Relatively Development Time is more	5) Relatively Development Time is less

7) Explain the differences between ASP and JSP?

The big difference between both of these technologies lies with the design of the software. JSP technology is server and platform independent whereas ASP relies primarily on Microsoft technologies.

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

Yes. Preemptive termination of request processing on an error condition is a good way to maximize the throughput of a high-volume JSP engine. The trick (assuming Java is your scripting language) is to use the return statement when we want to terminate further processing.

9) How to Protect JSPs from direct access ?

If the JSP is secured resource then we can place inside WEB-INF folder so that end user is not allowed to access directly by the name. We can provide the url pattern by configuring in web.xml

```
1)  <web-app>
2)  <servlet>
3)  <servlet-name>Demo JSP</servlet-name>
4)  <jsp-file>/WEB-INF/test.jsp</jsp-file>
5)  <sevlet>
6)  <servlet-mapping>
7)  <servlet-name>Demo JSP</servlet-name>
8)  <url-pattern>/test</url-pattern>
9)  </servlet-mapping>
10) ..
11) </web-app>
```

10) Explain JSP API ?

The JSP API contains only one package : javax.servlet.jsp

It contains the following 2 interfaces:

1) **JspPage:**
   This interface defines the two life cycle methods jspInit() and jspDestroy().

2) **HttpJspPage:**
   This interface defines only one life cyle method _jspService() method.
   Every generated servlet for the jsps should implement either JspPage or HttpJspPage interface either directly or indirectly.

11) What are the lifecycle phases of a JSP?

Life cycle of JSP contains the following phases:

- Page translation: -converting from .jsp file to .java file
- Page compilation: converting .java to .class file
- Page loading : This class file is loaded.
- Create an instance :- Instance of servlet is created
- jspInit() method is called
- _jspService() is called to handle service calls
- jspDestroy() is called to destroy it when the servlet is not required.

12) Explain the life-cycle mehtods in JSP?

The jspInit()- The container calls the jspInit() to initialize te servlet instance.It is called before any other method, and is called only once for a servlet instance.

The _jspservice()- The container calls the _jspservice() for each request, passing it the request and the response objects.

The jspDestroy()- The container calls this when it decides take the instance out of service. It is the last method called n the servlet instance.

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

JSP contains 3 life cycle methods namely jspInit( ), _jspService() and jspDestroy().
In these, jspInit() and jspDestroy() can be overridden and we cannot override _jspService().
Webcontainer always generate _jspService() method with JSP content. If we are writing _jspService() method , then generated servlet contains 2 _jspService() methods which will cause compile time error.
To show this difference _jspService() method is prefixed with '_' by the JSP container and the other two methods jspInit() and jspDestroy() has no special prefixes.

14) What is the jspInit() method?

The jspInit() method of the javax.servlet.jsp.JspPage interface is similar to the init() method of servlets. This method is invoked by the container only once when a JSP page is initialized. It can be overridden by a page author to initialize resources such as database and network connections, and to allow a JSP page to read persistent configuration data.

15) What is the _jspService() method?

The _jspService() method of the javax.servlet.jsp.HttpJspPage interface is invoked every time a new request comes to a JSP page. This method takes the HttpServletRequest and HttpServletResponse objects as its arguments. A page author cannot override this method, as its implementation is provided by the container.

16) What is the jspDestroy() method?

The jspDestroy() method of the javax.servlet.jsp.JspPage interface is invoked by the container when a JSP page is about to be destroyed. This method is similar to the destroy() method of servlets. It can be overridden by a page author to perform any cleanup operation such as closing a database connection.

17) What JSP lifecycle methods can I override?

We can override jspInit() and jspDestroy() methods but we cannot override _jspService() method.

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

By using JSP declation tag

```
1)  <%!
2)  public void jspInit() {
3)  . . .
4)  }
5)  %>
6)  <%!
7)  public void jspDestroy() {
8)  . . .
9)  }
10) %>
```

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

A java server page is executed within a Java container. A Java container converts a Java file into a servlet. Conversion happens only once when the application is deployed onto the web server. During the process of compilation Java compiler checks for modifications if any modifications are present it would modify and then execute it.

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

jspService() method will be written by the container hence any methods which are not to be overridden by the end user are typically written starting with an ''. This is the reason why we don't override _jspService() method in any JSP page.

21) How to pre-compile JSP?

Add jsp_precompile as a request parameter and send a request to the JSP file. This will make the jsp pre-compile.
http://localhost:8080/jsp1/test.jsp?jsp_precompile=true
It causes excution of JSP life cycle until jspInit() method without executing _jspService() method.

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

It removes the start-up lag that occurs when a container must translate a JSP page upon receipt of the first request.

23) How many JSP scripting elements and explain them?

Inside JSP 4 types of scripting elements are allowed.

1) Scriptlet <% any java code %>
   Can be used to place java code.

2) declarative <%! Java declaration %>
   Can be used to declare class level variables and methods

3) expression: <%= java expression %>
   To print java expressions in the JSP

4) comment <%-- jsp comment --%>

24) What is a Scriptlet?

JSP scriptlet can be used to place java code.

Syntax:

```
<%
    Any java code
%>
```

The java code present in the scriptlet will be placed directly inside _jspService() method

25) What is a JSP declarative?

JSP declarations are used to declare class variables and methods (both instance and static) in a JSP page. These declations will be placed directly at class level in the generated servlet and these are available to the entire JSP.

Syntax:

```
<%! This is my declarative %>
Eg: <%! int j = 10; %>
```

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

We can declare methods by using JSP declarative tag.

```
<%!
    public int add(inti, intj) {
                return i+j;
            }
%>
```

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

Variable declared inside declaration part is treated as a instance variable and will be placed directly at class level in the generated servlet.
<%! int k = 10; %>

Variable declared in a scriptlet will be placed inside _jspService() method of generated servlet.It acts as local variable.

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

28) What is a Expression?

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

Syntax:

<%= java expression %>

Eg: <%= new java.util.Date() %>

The expression in expression tag should not ends with semi-colon
The expression value will become argument to the out.pritln() method in the generated servlet

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

3 types of comments are allowed in JSP

JSP Comment:
<%-- this is jsp comment --%>
This is also known as hidden comment and it is visible only in the JSP and in rest of phases of JSP life cycle it is not visible.

HTML Comment:
<!-- this is HTMl comment -- >
This is also known as template text comment or output comment. It is visible in all phases of JSP including source code of generated response.

Java Comments:
With in the script lets we can use even java comments .

```
<%
// single line java comment
/* this is multiline comment */
%>
```

This type of comments also known as scripting comments and these are visible in the generated servlet also.

30) What is output comment?

The comment which is visible in the source of the response is called output comment.
<!-- this is HTMl comment -- >

31) What is a Hidden Comment?

<%-- this is jsp comment --%>
This is also known as JSP comment and it is visible only in the JSP and in rest of phases of JSP life cycle it is not visible.

32) How is scripting disabled?

- Scripting is disabled by setting the scripting-invalid element of the deployment descriptor to
- true. It is a subelement of jsp-property-group. Its valid values are true and false. The syntax
- for disabling scripting is as follows:

```
<jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <scripting-invalid>true</scripting-invalid>
</jsp-property-group>
```

33) What are the JSP implicit objects?

Implicit objects are by default available to the JSP. Being JSP author we can use these and not required to create it explicitly.

1) request
2) response
3) pageContext
4) session
5) application
6) out
7) config
8) page
9) exception

34) How does JSP handle run-time exceptions?

You can use the errorPage attribute of the page directive to have uncaught run-time exceptions automatically forwarded to an error processing page.

For example:
<%@ page errorPage="error.jsp" %> redirects the browser to the JSP page error.jsp if an uncaught exception is encountered during request processing.
Within error.jsp, if you indicate that it is an error-processing page, via the directive:
<%@ page isErrorPage="true" %>
In the error pages we can access exception implicit object.

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

You can make your JSPs thread-safe by having them implement the SingleThreadModel interface. This is done by adding the directive in the JSP.
<%@ page isThreadSafe="false" %>
The generated servlet can handle only one client request at time so that we can make JSP as thread safe. We can overcome data inconsistency problems by this approach.
The main limitation is it may affect the performance of the system.

36) What is the difference between ServletContext and PageContext?

ServletContext: Gives the information about the container and it represents an application.
PageContext: Gives the information about the Request and it can provide all other implicit JSP objects .

37) Is there a way to reference the "this" variable within a JSP page?

Yes, there is. The page implicit object is equivalent to "this", and returns a reference to the generated servlet.

38) Can you make use of a ServletOutputStream object from within a JSP page?

Yes . By using getOutputStream() method on response implicit object we can get it.

39) What is the page directive is used to prevent a JSP page from automatically creating a session?

session object is by default available to the JSP. We can make it unavailable by using page directive as follows.
<%@ page session="false">

40) What's a better approach for enabling thread-safe servlets and JSPs? Single Thread Model Interface OR Synchronization?

Synchronized keyword is recommended to use to get thread-safety.

41) What are various attributes Of Page Directive?

Page directive contains the following 13 attributes.

1) language
2) extends
3) import
4) session
5) isThreadSafe
6) info
7) errorPage

8) isError page
9) contentType
10) isELIgnored
11) buffer
12) autoFlush
13) pageEncoding

42) Explain about autoflush?

This command is used to autoflush the contents. If a value of true is used it indicates to flush the buffer whenever it is full. In case of false it indicates that an exception should be thrown whenever the buffer is full. If you are trying to access the page at the time of conversion of a JSP into servlet will result in error.

43) How do you restrict page errors display in the JSP page?

You first set "errorPage" attribute of PAGE directive to the name of the error page (ie errorPage="error.jsp")in your jsp page .
Then in the error.jsp page set "isErrorpage=TRUE".
When an error occur in your jsp page, then the control will be automatically forward to error page.

44) What are the different scopes available fos JSPs ?

There are 4 types of scopes are allowed in the JSP.

1) page - with in the same page
2) request - after forward or include also you will get the request scope data.
3) session - after senRedirect also you will get the session scope data. All data stored in session is available to end user till session closed or browser closed.
4) application - Data will be available throughout the application. One user can store data in application scope and other can get the data from application scope.

45) When do use application scope?

If we want to make our data available to the entire application then we have to use application scope.

46) What are the different scope valiues for the <jsp:useBean>?

The different scope values for <jsp:useBean> are
1) page
2) request
3) session
4) application

47) How do I use a scriptlet to initialize a newly instantiated bean?

jsp:useBean action may optionally have a body. If the body is specified, its contents will be automatically invoked when the specified bean is instantiated. Typically, the body will contain scriptlets or jsp:setProperty tags to initialize the newly instantiated bean, although you are not restricted to using those alone.
The following example shows the "today" property of the Foo bean initialized to the current date when it is instantiated. Note that here, we make use of a JSP expression within the jsp:setProperty action.

```
<jsp:useBean id="foo" class="com.Bar.Foo" >
        <jsp:setProperty name="foo" property="x"
      value="<%=java.text.DateFormat.getDateInstance().format(new java.util.Date()) %>" / >
            <%-- scriptlets calling bean setter methods go here --%>
</jsp:useBean >
```

48) Can a JSP page instantiate a serialized bean?

No problem! The use Bean action specifies the beanName attribute, which can be used for indicating a serialized bean.
For example:
A couple of important points to note. Although you would have to name your serialized file "filename.ser", you only indicate "filename" as the value for the beanName attribute. Also, you will have to place your serialized file within the WEB-INF/jspbeans directory for it to be located by the JSP engine.

49) How do we include static files within a jsp page ?

We can include static files in JSP by using include directive (static include)
<%@ include file="header.jsp" %>
The content of the header.jsp will be included in the current jsp at translation time. Hence this inclusion is also known as static include.

50) In JSPs how many ways are possible to perform inclusion?

In JSP, we can perform inclusion in the following ways.

1) By include directive:

   <%@ include file="header.jsp" %>
   The content of the header.jsp will be included in the current jsp at translation time. Hence this inclusion is also known as static include.

2) By include action:

   <jsp:include page="header.jsp" />
   The response of the jsp will be included in the current page response at request processing time(run time) hence it is also known as dynamic include.

3) By using pageContext implicit object

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

This inclusion also happened at request processing time(run time).

4) By using RequestDispatcher object

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

51) In which situation we can use static include and dynamic include in JSPs?

If the target resource (included resource) won't change frequently, then it is recommended to use static include.
<%@ include file="header.jsp" %>
If the target resource(Included page) will change frequently , then it is recommended to use dynamic include.
< jsp:include page="header.jsp" />

OCWCD

Question Bank

Unit 1: The Java Server Pages (JSP) Technology Model

1. Identify, describe, or write the JSP code for the following elements: (a) template text, (b) scripting elements (comments, directives, declarations, scriptlets, and expressions), (c) standard and custom actions, and (d) expression language elements.
2. Write JSP code that uses the directives: (a) 'page' (with attributes 'import', 'session', 'contentType', and 'isELIgnored'), (b) 'include', and (c) 'taglib'.
3. Write a JSP Document (XML-based document) that uses the correct syntax.
4. Describe the purpose and event sequence of the JSP page life cycle: (1) JSP page translation, (2) JSP page compilation, (3) load class, (4) create instance, (5) call the jspInit method, (6) call the _jspService method, and (7) call the jspDestroy method.
5. Given a design goal, write JSP code using the appropriate implicit objects: (a) request, (b) response, (c) out, (d) session, (e) config, (f) application, (g) page, (h) pageContext, and (i) exception.
6. Configure the deployment descriptor to declare one or more tag libraries, deactivate the evaluation language, and deactivate the scripting language. 6.7 Given a specific design goal for including a JSP segment in another page, write the JSP code that uses the most appropriate inclusion mechanism (the include directive or the jsp:include standard action).

Q1. To take advantage of the capabilities of modern browsers that use web standards, such as XHTML and CSS, your web application is being converted from simple JSP pages to JSP Document format. However, one of your JSPs, /scripts/screenFunctions.jsp, generates a JavaScript file. This file is included in several web forms to create screen-specific validation functions and are included in these pages with the following statement:

```
<head>
<script src='/scripts/screenFunctions.jsp'
language='javascript'
type='application/javascript'> </script>
</head>
<!-- body of the web form -->
```

Which JSP code snippet declares that this JSP Document is a JavaScript file?

A. <%@ page contentType='application/javascript' %>
B. <jsp:page contentType='application/javascript' />
C. <jsp:document contentType='application/javascript' />
D. <jsp:directive.page contentType='application/javascript' />
E. No declaration is needed because the web form XHTML page already declares the MIME type of the /scripts/screenFunctions.jsp file in the <script> tag.

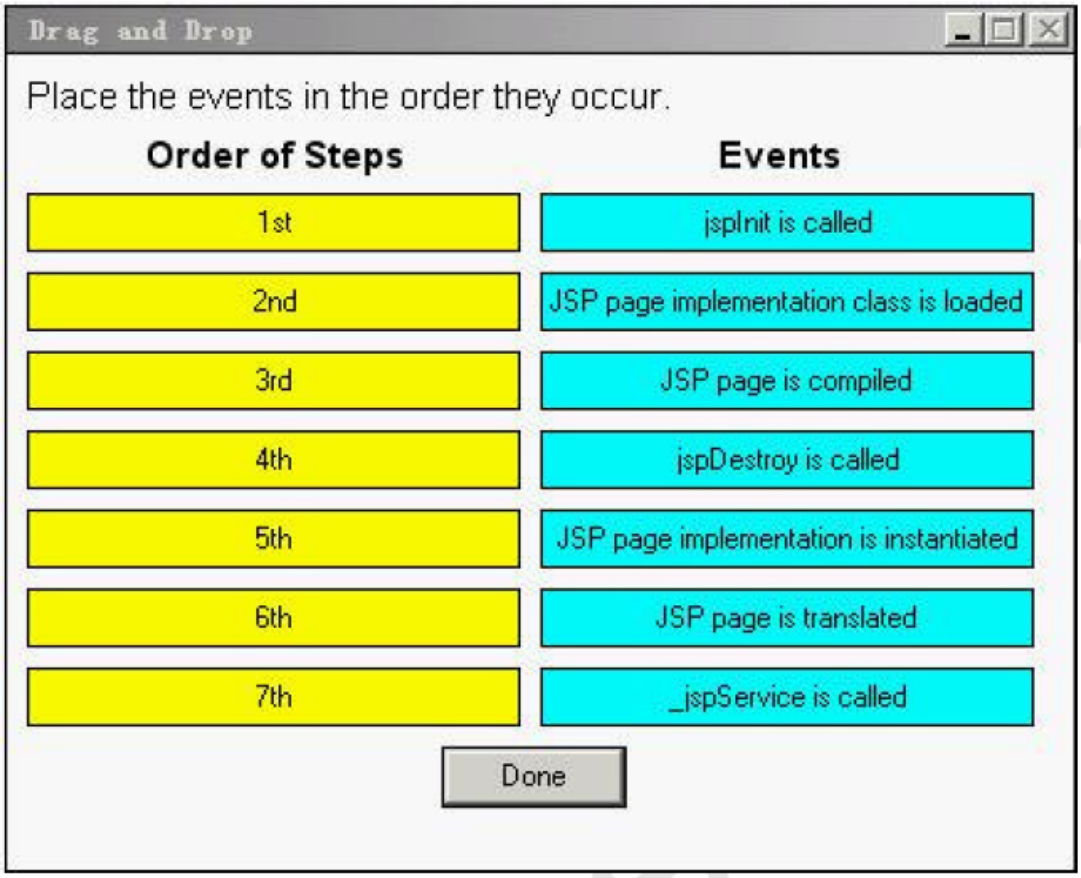
Answer: D

Q2. Which implicit object is used in a JSP page to retrieve values associated with <context-param> entries in the deployment descriptor?
A. config
B. request
C. session

D. application

Answer: D

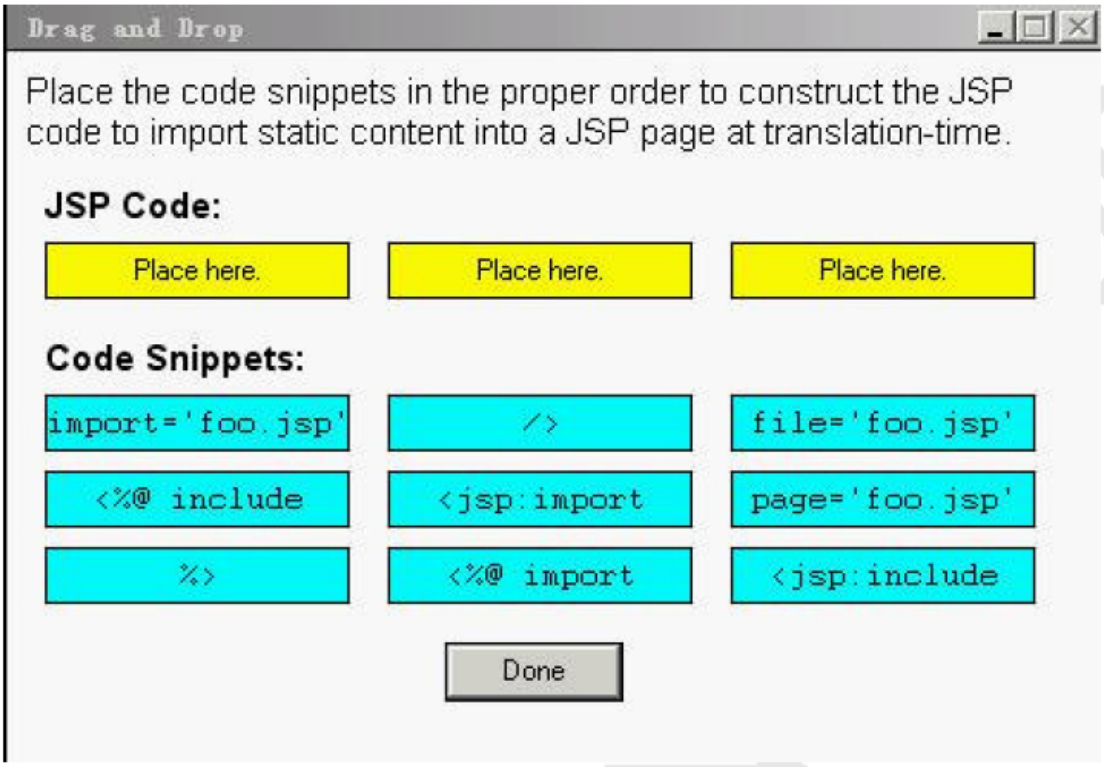
Q3. Click the Task button.
Place the events in the order they occur.



Answer:

JSP page is translated
JSP page is compiled
JSP page implementation class is loaded
jspInit is called
_jspService is called
jspDestroy is calle

Q4. Click the Task button.
Place the code snippets in the proper order to construct the JSP code to import static content into a JSP page at translation-time.



Answer :
<%@ include file='foo.jsp' %.

Q5. You have created a JSP that includes instance variables and a great deal of scriptlet code. Unfortunately, after extensive load testing, you have discovered several race conditions in your JSP scriptlet code. To fix these problems would require significant recoding, but you are already behind schedule. Which JSP code snippet can you use to resolve these concurrency problems?

A. <%@ page isThreadSafe='false' %>
B. <%@ implements SingleThreadModel %>
C. <%! implements SingleThreadModel %>
D. <%@ page useSingleThreadModel='true' %>
E. <%@ page implements='SingleThreadModel' %>
Answer: A

Q6. For debugging purposes, you need to record how many times a given JSP is invoked before the user's session has been created. The JSP's destroy method stores this information to a database. Which JSP code snippet keeps track of this count for the lifetime of the JSP page?
A. <%! int count = 0; %>
<% if (request.getSession(false) == null) count++; %>
B. <%@ int count = 0; %>
<% if (request.getSession(false) == null) count++; %>
C. <% int count = 0;
if (request.getSession(false) == null) count++; %>
D. <%@ int count = 0;
if (request.getSession(false) == null) count++; %>

E. <%! int count = 0;
if (request.getSession(false) == null) count++; %>
Answer: A

Q7. For manageability purposes, you have been told to add a "count" instance variable to a critical JSP Document so that a JMX MBean can track how frequent this JSP is being invoked. Which JSP code snippet must you use to declare this instance variable in the JSP Document?
A. <jsp:declaration>
int count = 0;
<jsp:declaration>
B. <%! int count = 0; %>
C. <jsp:declaration.instance>
int count = 0;
<jsp:declaration.instance>
D. <jsp:scriptlet.declaration>
int count = 0;
<jsp:scriptlet.declaration>
Answer: A

Q8. In a JSP-centric web application, you need to create a catalog browsing JSP page. The catalog is stored as a List object in the catalog attribute of the webapp's ServletContext object.
Which scriptlet code snippet gives you access to the catalog object?
A. <% List catalog = config.getAttribute("catalog"); %>
B. <% List catalog = context.getAttribute("catalog"); %>
C. <% List catalog = application.getAttribute("catalog"); %>
D. <% List catalog = servletContext.getAttribute("catalog"); %>
Answer: C

Q9. Given the element from the web application deployment descriptor:
<jsp-property-group>
<url-pattern>/main/page1.jsp</url-pattern>
<scripting-invalid>true</scripting-invalid>
</jsp-property-group>
and given that /main/page1.jsp contains:
<% int i = 12; %>
<%= i %>
What is the result?
A.
B. 12
C. The JSP fails to execute.
D. <% int i = 12 %>
<%= i %>
Answer: C

Q10. You are creating a new JSP page and you need to execute some code that acts when the page is first executed, but only once. Which three are possible mechanisms for performing this initialization code?(Choose hree.)
A. In the init method.
B. In the jspInit method.
C. In the constructor of the JSP's Java code.

D. In a JSP declaration, which includes an initializer block.
E. In a JSP declaration, which includes a static initializer block.
Answer: B, D, E

Q11. You are writing a JSP that includes scriptlet code to declare a List variable and initializes that variable to an ArrayList object. Which two JSP code snippets can you use to import these list types? (Choose two.)
A. <%! import java.util.*; %>
B. <%! import java.util.List;
import java.util.ArrayList; %>
C. <%@ page import='java.util.List'
import='java.util.ArrayList' %>
D. <%@ import types='java.util.List'
types='java.util.ArrayList' %>
E. <%@ page import='java.util.List,java.util.ArrayList' %>
F. <%@ import types='java.util.List,java.util.ArrayList' %>
Answer: C, E

Q12. Every page of your web site must include a common set of navigation menus at the top of the page. This menu is static HTML and changes frequently, so you have decided to use JSP's static import mechanism.
Which JSP code snippet accomplishes this goal?
A. <%@ import file='/common/menu.html' %>
B. <%@ page import='/common/menu.html' %>
C. <%@ import page='/common/menu.html' %>
D. <%@ include file='/common/menu.html' %>
E. <%@ page include='/common/menu.html' %>
F. <%@ include page='/common/menu.html' %>
Answer: D

Q13. For manageability purposes, you have been told to add a "count" instance variable to a critical JSP Document so that a JMX MBean can track how frequent this JSP is being invoked.
Which JSP code snippet must you use to declare this instance variable in the JSP Document?
A. <jsp:declaration>
int count = 0;
<jsp:declaration>
B. <%! int count = 0; %>
C. <jsp:declaration.instance>
int count = 0;
<jsp:declaration.instance>
D. <jsp:scriptlet.declaration>
int count = 0;
<jsp:scriptlet.declaration>
Answer: A

Q14. You are creating a new JSP page and you need to execute some code that acts when the page is first executed, but only once. Which three are possible mechanisms for performing this initialization code?
(Choose three.)

A. In the init method.
B. In the jspInit method.
C. In the constructor of the JSP's Java code.
D. In a JSP declaration, which includes an initializer block.
E. In a JSP declaration, which includes a static initializer block.

Answer: B, D, E

Q15. The JSP developer wants a comment to be visible in the final output to the browser. Which comment style needs to be used in a JSP page?
A. <!-- this is a comment -->
B. <% // this is a comment %>
C. <%-- this is a comment --%>
D. <% /** this is a comment **/ %>
Answer: A

Q16. Which is a benefit of precompiling a JSP page?
A. It avoids initialization on the first request.
B. It provides the ability to debug runtime errors in the application.
C. It provides better performance on the first request for the JSP page.
D. It avoids execution of the _jspService method on the first request.
Answer: C

Q17. In a JSP-centric web application, you need to create a catalog browsing JSP page. The catalog is stored as a List object in the catalog attribute of the webapp's ServletContext object.
Which scriptlet code snippet gives you access to the catalog object?
A. <% List catalog = config.getAttribute("catalog"); %>
B. <% List catalog = context.getAttribute("catalog"); %>
C. <% List catalog = application.getAttribute("catalog"); %>
D. <% List catalog = servletContext.getAttribute("catalog"); %>
Answer: C

Q18. Which ensures that a JSP response is of type "text/plain"?
A. <%@ page mimeType="text/plain" %>
B. <%@ page contentType="text/plain" %>
C. <%@ page pageEncoding="text/plain" %>
D. <%@ page contentEncoding="text/plain" %>
E. <% response.setEncoding("text/plain"); %>
F. <% response.setMimeType("text/plain"); %>
Answer: B

Unit 2: Building JSP Pages Using Standard Actions

- Given a design goal, create a code snippet using the following standard actions: jsp:useBean (with attributes: 'id', 'scope', 'type', and 'class'), jsp:getProperty, jsp:setProperty (with all attribute combinations), and jsp:attribute.

- Given a design goal, create a code snippet using the following standard actions: jsp:include, jsp:forward, and jsp:param

Q1. Given the JSP code:
10. <html>
11. <body>
12. <jsp:useBean id='customer' class='com.example.Customer' />
13. Hello, ${customer.title} ${customer.lastName}, welcome
14. to Squeaky Beans, Inc.
15. </body>
16. </html>

Which three types of JSP code are used? (Choose three.)

A. Java code
B. template text
C. scripting code
D. standard action
E. expression language

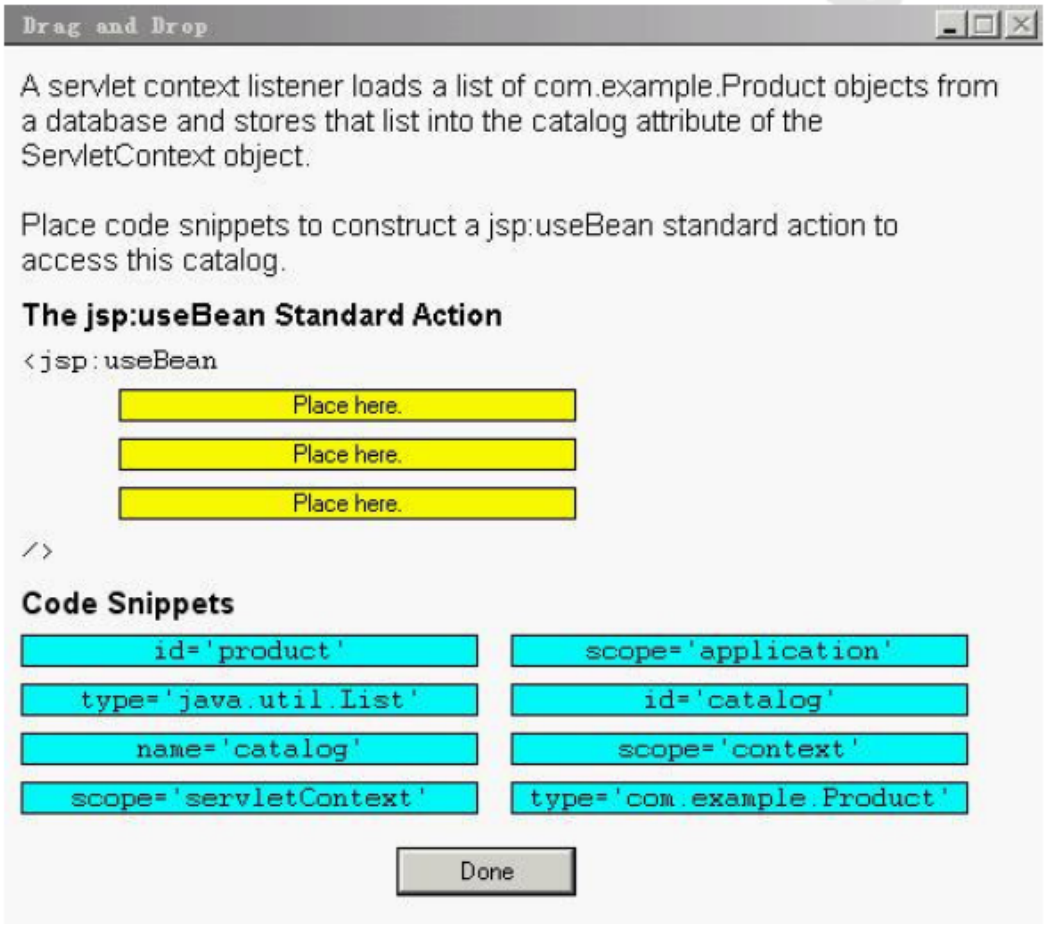
Answer: B, D, E

Q2. Which JSP standard action can be used to import content from a resource called foo.jsp?

A. <jsp:import file='foo.jsp' />
B. <jsp:import page='foo.jsp' />
C. <jsp:include page='foo.jsp' />
D. <jsp:include file='foo.jsp' />
E. <jsp:import>foo.jsp</jsp:import>
F. <jsp:include>foo.jsp</jsp:include>

Answer: C

Q3. Click the Task button.
A servlet context listener loads a list of com.example.Product objects from a database and stores that list into the catalog attribute of the ServletContext object.Place code snippets to construct a jsp:useBean standard action to access this catalog.



Answer:
<jsp:useBean id='catalog' type='java.util.List' scope='application' />

Q4. A session-scoped attribute is stored by a servlet, and then that servlet forwards to a JSP page. Which three jsp:useBean attributes must be used to access this attribute in the JSP page? (Choose three.)

A. id **B. name**
C. bean **D. type**
E. scope **F. beanName**

Answer: A, D, E

Q5. You need to create a JavaBean object that is used only within the current JSP page. It must NOT be accessible to any other page including those that this page might import. Which JSP standard action can accomplish this goal?

A. <jsp:useBean id='pageBean' type='com.example.MyBean' />
B. <jsp:useBean id='pageBean' class='com.example.MyBean' />
C. <jsp:makeBean id='pageBean' type='com.example.MyBean' />
D. <jsp:makeBean id='pageBean' class='com.example.MyBean' />
E. <jsp:useBean name='pageBean' class='com.example.MyBean' />
F. <jsp:makeBean name='pageBean' class='com.example.MyBean' />
Answer: B

Q6. Assume a JavaBean com.example.GradedTestBean exists and has two attributes. The attribute name is of type java.lang.String and the attribute score is of type java.lang.Integer.
An array of com.example.GradedTestBean objects is exposed to the page in a request-scoped attribute called results. Additionally, an empty java.util.HashMap called resultMap is placed in the page scope.
A JSP page needs to add the first entry in results to resultMap, storing the name attribute of the bean as the key and the score attribute of the bean as the value.

Which code snippet of JSTL code satisfies this requirement?

A. ${resultMap[results[0].name] = results[0].score}
B. <c:set var="${resultMap}" key="${results[0].name}"
value="${results[0].score}" />
C. <c:set var="resultMap" property="${results[0].name}">
${results[0].value}
</c:set>
D. <c:set var="resultMap" property="${results[0].name}"
value="${results[0].score}" />
E. <c:set target="${resultMap}" property="${results[0].name}"
value="${results[0].score}" />

Answer: E

Q7. A JSP page needs to set the property of a given JavaBean to a value that is calculated with the JSP page. Which three jsp:setProperty attributes must be used to perform this initialization? (Choose three.)

A. id
B. val
C. name
D. param
E. value
F. property
G. attribute

Answer: C, E, F

Q8. Your web application views all have the same header, which includes the <title> tag in the <head> element of the rendered HTML. You have decided to remove this redundant HTML code from your JSPs and put it into a single JSP called /WEB-INF/jsp/header.jsp. However, the title of each page is unique, so you have decided to use a variable called pageTitle to parameterize this in the header JSP, like this:

10. <title>${param.pageTitle}<title>

Which JSP code snippet should you use in your main view JSPs to insert the header and pass the pageTitle variable?

A. <jsp:insert page='/WEB-INF/jsp/header.jsp'>
${pageTitle='Welcome Page'}
</jsp:insert>
B. <jsp:include page='/WEB-INF/jsp/header.jsp'>

${pageTitle='Welcome Page'}
</jsp:include>
C. <jsp:include file='/WEB-INF/jsp/header.jsp'>
${pageTitle='Welcome Page'}
</jsp:include>
D. <jsp:insert page='/WEB-INF/jsp/header.jsp'>
<jsp:param name='pageTitle' value='Welcome Page' />
</jsp:insert>
E. <jsp:include page='/WEB-INF/jsp/header.jsp'>
<jsp:param name='pageTitle' value='Welcome Page' />
</jsp:include>

Answer: E

Q9. A JSP page needs to instantiate a JavaBean to be used by only that page.Which two jsp:useBean attributes must be used to access this attribute in the JSP page? (Choose two.)

A. id
B. type
C. name
D. class
E. scope
F. create

Answer: A, D

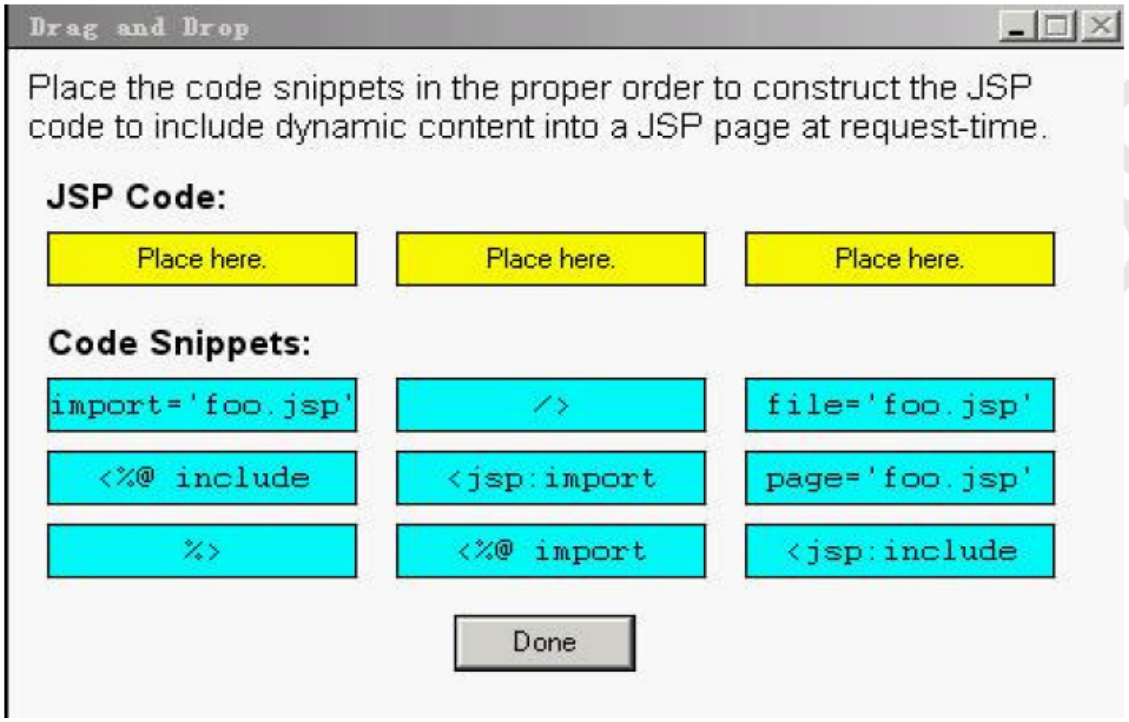
Q10. Click the Exhibit button
Given the HTML form:
1. <html>
2. <body>
3. <form action="submit.jsp">
4. Name: <input type="text" name="i1">
5. Price: <input type="text" name="i2">
6. <input type="submit">
7. </form>
8. </body>
9. </html>

Assume the product attribute does NOT yet exist in any scope.
Which code snippet, in submit.jsp, instantiates an instance of com.example.Product that contains the results of the form submission?

```
1. package com.example;
2.
3. public class Product {
4.   private String name;
5.   private double price;
6.
7.   public Product() {
8.     this( "Default", 0.0 );
9.   }
10.
11.  public Product( String name, double
price ) {
12.    this.name = name;
13.    this.price = price;
14.  }
15.
16.  public String getName() {
17.    return name;
18.  }
19.
20.  public void setName(String name) {
21.    this.name = name;
22.  }
23.
24.  public double getPrice() {
25.    return price;
26.  }
27.
28.  public void setPrice(double price) {
29.    this.price = price;
30.  }
31. }
```

A. <jsp:useBean id="com.example.Product" />
<jsp:setProperty name="product" property="*" />
B. <jsp:useBean id="product" class="com.example.Product" />
${product.name = param.i1}
${product.price = param.i2}
C. <jsp:useBean id="product" class="com.example.Product">
<jsp:setProperty name="product" property="name"
param="i1" />
<jsp:setProperty name="product" property="price"
param="i2" />
</jsp:useBean>
D. <jsp:useBean id="product" type="com.example.Product">
<jsp:setProperty name="product" property="name"
value="<%= request.getParameter("i1") %>" />
<jsp:setProperty name="product" property="price"
value="<%= request.getParameter("i2") %>" />
</jsp:useBean>
Answer: C

Q11. Click the Task button.
Place the code snippets in the proper order to construct the JSP code to include dynamic content into a JSP page at request-time.



Ans:
<jsp:include page='foo.jsp' />

Q12. Click the Exhibit button.
Given:
10. <form action='create_product.jsp'>
**11. Product Name: <input type='text' name='prodName'/>
**
**12. Product Price: <input type='text' name='prodPrice'/>
**
13. </form>
For a given product instance, which three jsp:setProperty attributes must be used to initialize its properties from the HTML form? (Choose three.)

```
1. package com.example;
2.
3. public class Product {
4.   private String name;
5.   private double price;
6.
7.   public Product() {
8.     this( "Default", 0.0 );
9.   }
10.
11.  public Product( String name, double
price ) {
12.    this.name = name;
13.    this.price = price;
14.  }
15.
16.  public String getName() {
17.    return name;
18.  }
19.
20.  public void setName(String name) {
21.    this.name = name;
22.  }
23.
24.  public double getPrice() {
25.    return price;
26.  }
27.
28.  public void setPrice(double price) {
29.    this.price = price;
30.  }
31. }
```

A. id
B. name
C. type
D. param
E. property
F. reqParam
G. attribute
Answer: B, D, E

Q13. Given:
1. package com.example;
2.
3. public abstract class AbstractItem {
4. private String name;
...

13. }

Assume a concrete class com.example.ConcreteItem extends com.example. AbstractItem. A servlet sets a session-scoped attribute called "item" that is an instance of com.example.ConcreteItem and then forwards to a JSP page.
Which two are valid standard action invocations that expose a scripting variable to the JSP page?(Choose two.)
A. <jsp:useBean id="com.example.ConcreteItem"
scope="session" />
B. <jsp:useBean id="item" type="com.example.ConcreteItem"
scope="session" />
C. <jsp:useBean id="item" class="com.example.ConcreteItem"
scope="session" />
D. <jsp:useBean id="item" type="com.example.ConcreteItem"
class="com.example.AbstractItem"
scope="session" />

Answer: B, C

Q14. Your web application views all have the same header, which includes the <title> tag in the <head> element of the rendered HTML. You have decided to remove this redundant HTML code from your JSPs and put it into a single JSP called /WEB-INF/jsp/header.jsp. However, the title of each page is unique, so you have decided to use a variable called pageTitle to parameterize this in the header JSP, like this:

10. <title>${param.pageTitle}<title>

Which JSP code snippet should you use in your main view JSPs to insert the header and pass the pageTitle variable?

A. <jsp:insert page='/WEB-INF/jsp/header.jsp'>
${pageTitle='Welcome Page'}
</jsp:insert>
B. <jsp:include page='/WEB-INF/jsp/header.jsp'>
${pageTitle='Welcome Page'}
</jsp:include>
C. <jsp:include file='/WEB-INF/jsp/header.jsp'>
${pageTitle='Welcome Page'}
</jsp:include>
D. <jsp:insert page='/WEB-INF/jsp/header.jsp'>
<jsp:param name='pageTitle' value='Welcome Page' />
</jsp:insert>
E. <jsp:include page='/WEB-INF/jsp/header.jsp'>
<jsp:param name='pageTitle' value='Welcome Page' />
</jsp:include>
Answer: E

Q15. Click the Exhibit button.

Given the JSP code:
1. <%

2. pageContext.setAttribute("product",
3. new com.example.Product("Pizza", 0.99));
4. %>
5. <%-- insert code here --%>
Which two, inserted at line 5, output the name of the product in the response? (Choose two.)

```
1. package com.example;
2.
3. public class Product {
4.   private String name;
5.   private double price;
6.
7.   public Product() {
8.     this( "Default", 0.0 );
9.   }
10.
11.  public Product( String name, double
price ) {
12.    this.name = name;
13.    this.price = price;
14.  }
15.
16.  public String getName() {
17.    return name;
18.  }
19.
20.  public void setName(String name) {
21.    this.name = name;
22.  }
23.
24.  public double getPrice() {
25.    return price;
26.  }
27.
28.  public void setPrice(double price) {
29.    this.price = price;
30.  }
31. }
```

A. <%= product.getName() %>
B. <jsp:useBean id="product" class="com.example.Product" />
<%= product.getName() %>
C. <jsp:useBean id="com.example.Product" scope="page">
<%= product.getName() %>
</jsp:useBean>
D. <jsp:useBean id="product" type="com.example.Product"
scope="page" />
<%= product.getName() %>
E. <jsp:useBean id="product" type="com.example.Product">
<%= product.getName() %>

</jsp:useBean>

Q16.In a JSP-centric shopping cart application, you need to move a client's home address of the Customer object into the shipping address of the Order object. The address data is stored in a value object class called Address with properties for: street address, city, province, country, and postal code. Which two JSP code snippets can be used to accomplish this goal? (Choose two.)

A. <c:set var='order' property='shipAddress'
value='${client.homeAddress}' />

B. <c:set target='${order}' property='shipAddress'
value='${client.homeAddress}' />

C. <jsp:setProperty name='${order}' property='shipAddress'
value='${client.homeAddress}' />

D. <c:set var='order' property='shipAddress'>
<jsp:getProperty name='client' property='homeAddress' />
</c:store>

E. <c:set target='${order}' property='shipAddress'>
<jsp:getProperty name='client' property='homeAddress' />
</c:set>

F. <c:setProperty name='${order}' property='shipAddress'>
<jsp:getProperty name='client' property='homeAddress' />
</c:setProperty>

Answer: B, E

Q17. Click the Exhibit button.

Assume the product attribute does NOT yet exist in any scope.
Which two create an instance of com.example.Product and initialize the name and price properties to the name and price request parameters? (Choose two.)

```
1. package com.example;
2.
3. public class Product {
4.   private String name;
5.   private double price;
6.
7.   public Product() {
8.     this( "Default", 0.0 );
9.   }
10.
11.  public Product( String name, double
price ) {
12.    this.name = name;
13.    this.price = price;
14.  }
15.
16.  public String getName() {
17.    return name;
18.  }
19.
20.  public void setName(String name) {
21.    this.name = name;
22.  }
23.
24.  public double getPrice() {
25.    return price;
26.  }
27.
28.  public void setPrice(double price) {
29.    this.price = price;
30.  }
31. }
```

A. <jsp:useBean id="product" class="com.example.Product" />
<jsp:setProperty name="product" property="*" />
B. <jsp:useBean id="product" class="com.example.Product" />
<% product.setName(request.getParameter("name")); %>
<% product.setPrice(request.getParameter("price")); %>
C. <jsp:useBean id="product" class="com.example.Product" />
<jsp:setProperty name="product" property="name"
value="${param.name}" />
<jsp:setProperty name="product" property="price"
value="${param.price}" />
D. <jsp:useBean id="product" class="com.example.Product">
<jsp:setProperty name="product" property="name"
value="${name}" />
<jsp:setProperty name="product" property="price"
value="${price}" />
</jsp:useBean>

Answer: A, C

Q18. Click the Exhibit button.
A session-scoped attribute, product, is stored by a servlet. That servlet then forwards to a JSP page. This attribute holds an instance of the com.example.Product class with a name property of "The Matrix" and price property of 39.95.

Given the JSP page code snippet:

1. <jsp:useBean id='product' class='com.example.Product'>
2. <jsp:setProperty name='product' property='price' value='49.95'/>
3. </jsp:useBean>
4. <%= product.getName() %> costs <%= product.getPrice() %>
What is the response output of this JSP page code snippet?

```
1. package com.example;
2.
3. public class Product {
4.    private String name;
5.    private double price;
6.
7.    public Product() {
8.       this( "Default", 0.0 );
9.    }
10.
11.   public Product( String name, double
price ) {
12.      this.name = name;
13.      this.price = price;
14.   }
15.
16.   public String getName() {
17.      return name;
18.   }
19.
20.   public void setName(String name) {
21.      this.name = name;
22.   }
23.
24.   public double getPrice() {
25.      return price;
26.   }
27.
28.   public void setPrice(double price) {
29.      this.price = price;
30.   }
31. }
```

A. Default costs 0.0
B. Default costs 49.95
C. Default costs 39.95
D. The Matrix costs 0.0
E. The Matrix costs 49.95
F. The Matrix costs 39.95

Answer: B

Q19. Click the Exhibit button.
A servlet sets a session-scoped attribute product with an instance of com.example.Product and forwards to a JSP.
Which two output the name of the product in the response? (Choose two.)

```
1. package com.example;
2.
3. public class Product {
4.   private String name;
5.   private double price;
6.
7.   public Product() {
8.     this( "Default", 0.0 );
9.   }
10.
11.  public Product( String name, double
price ) {
12.    this.name = name;
13.    this.price = price;
14.  }
15.
16.  public String getName() {
17.    return name;
18.  }
19.
20.  public void setName(String name) {
21.    this.name = name;
22.  }
23.
24.  public double getPrice() {
25.    return price;
26.  }
27.
28.  public void setPrice(double price) {
29.    this.price = price;
30.  }
31. }
```

A. ${product.name}
B. <jsp:getProperty name="product" property="name" />
C. <jsp:useBean id="com.example.Product" />
<%= product.getName() %>
D. <jsp:getProperty name="product" class="com.example.Product"
property="name" />
E. <jsp:useBean id="product" type="com.example.Product">
<%= product.getName() %>
</jsp:useBean>

Answer: A, B

Q20. You are building your own layout mechanism by including dynamic content for the page's header and footer sections. The footer is always static, but the header generates the <title> tag that requires the page name to be specified dynamically when the header is imported. Which JSP code snippet performs the import of the header content?

A. <jsp:include page='/WEB-INF/jsp/header.jsp'>
<jsp:param name='pageName' value='Welcome Page' />
www.TestsNow.com

- 140 -

</jsp:include>
B. <jsp:import page='/WEB-INF/jsp/header.jsp'>
<jsp:param name='pageName' value='Welcome Page' />
</jsp:import>
C. <jsp:include page='/WEB-INF/jsp/header.jsp'>
<jsp:attribute name='pageName' value='Welcome Page' />
</jsp:include>
D. <jsp:import page='/WEB-INF/jsp/header.jsp'>
<jsp:attribute name='pageName' value='Welcome Page' />
</jsp:import>

Answer: A

Unit 3: Building JSP Pages Using the Expression Language (EL)

- Given a scenario, write EL code that accesses the following implicit variables including pageScope, requestScope, sessionScope, and applicationScope, param and paramValues, header and headerValues, cookie, initParam and pageContext.
- Given a scenario, write EL code that uses the following operators: property access (the . operator), collection access (the [] operator).

Q1. Which EL expression evaluates to the request URI?
A. ${requestURI}
B. ${request.URI}
C. ${request.getURI}
D. ${request.requestURI}
E. ${requestScope.requestURI}
F. ${pageContext.request.requestURI}
G. ${requestScope.request.requestURI}
Answer: F

Q2. Given:
```
1. <% int[] nums = {42,420,4200};
2. request.setAttribute("foo", nums); %>
3. ${5 + 3 lt 6}
4. ${requestScope['foo'][0] ne 10 div 0}
5. ${10 div 0}
```
What is the result?
A. true true
B. false true
C. false true 0
D. true true Infinity
E. false true Infinity
F. An exception is thrown.
G. Compilation or translation fails.
Answer: E

Q3. You have created a web application that you license to real estate brokers.The webapp is highly customizable including the email address of the broker, which is placed on the footer of each page. This is configured as a context parameter in the deployment descriptor:

```
10. <context-param>
11. <param-name>footerEmail</param-name>
12. <param-value>joe@estates-r-us.biz</param-value>
13. </context-param>
```

Which EL code snippet will insert this context parameter into the footer?
A. Contact me
B. Contact me
C. Contact me
D. Contact me
E. Contact me
Answer: C

Q4. Given an EL function foo, in namespace func, that requires a long as a parameter and returns a Map, which two are valid invocations of function foo? (Choose two.)
A. ${func(1)} **B. ${foo:func(4)}**
C. ${func:foo(2)} **D. ${foo(5):func}**
E. ${func:foo("easy")} **F. ${func:foo("3").name}**

Answer: C, F

Q5. Click the Exhibit button.
The Appliance class is a Singleton that loads a set of properties into a Map from an external data source.Assume:
An instance of the Appliance class exists in the application-scoped attribute, appl
The appliance object includes the name property that maps to the value Cobia.
The request-scoped attribute, prop, has the value name.

Which two EL code snippets will display the string Cobia? (Choose two.)

```
 1. package com.example;
 2. import java.util.*;
 3. public class Appliance {
 4.    private Map<String,String> props;
 5.    public Appliance() {
 6.      this.props = new
HashMap<String,String();
 7.      initialize();
 8.    }
 9.    public Map<String,String>
getProperties() {
10.      return this.props;
11.    }
12.    private void initialize() {
13.      // code to load appliance properties
14.    }
15. }
```

A. ${appl.properties.name}
B. ${appl.properties.prop}
C. ${appl.properties[prop]}
D. ${appl.properties[name]}
E. ${appl.getProperties().get(prop)}
F. ${appl.getProperties().get('name')}
Answer: A, C

Q6.Given:
11. <%
12. request.setAttribute("vals", new String[]{"1","2","3","4"});
13. request.setAttribute("index", "2");
14. %>
15. <%-- insert code here --%>

Which three EL expressions, inserted at line 15, are valid and evaluate to "3"? (Choose three.)
A. ${vals.2}
B. ${vals["2"]}
C. ${vals.index}
D. ${vals[index]}
E. ${vals}[index]
F. ${vals.(vals.index)}
G. ${vals[vals[index-1]]}
Answer: B, D, G

Q7. Given:
11. <% java.util.Map map = new java.util.HashMap();
12. request.setAttribute("map", map);
13. map.put("a", "true");
14. map.put("b", "false");
15. map.put("c", "42"); %>

Which three EL expressions are valid and evaluate to true? (Choose three.)

A. ${not map.c}
B. ${map.d or map.a}
C. ${map.a and map.d}
D. ${map.false or map.true}
E. ${map.a and map.b or map.a}
F. ${map['true'] or map['false']}

Answer: A, B, E

Q8. Given:
http://com.example/myServlet.jsp?num=one&num=two&num=three
Which two produce the output "one, two and three"? (Choose two.)
A. ${param.num[0],[1] and [2]}
B. ${paramValues[0],[1] and [2]}
C. ${param.num[0]}, ${param.num[1]} and ${param.num[2]}
D. ${paramValues.num[0]}, ${paramValues.num[1]} and ${paramValues.num[2]}
E. ${paramValues["num"][0]}, ${paramValues["num"][1]} and ${paramValues["num"][2]}
F. ${parameterValues.num[0]}, ${parameterValues.num[1]} and ${parameterValues.num[2]}
G. ${parameterValues["num"]["0"]}, ${parameterValues["num"]["1"]} and
${parameterValues["num"]["2"]}

Answer: D, E

Q9. Given a web application in which the cookie userName is expected to
contain the name of the user. Which EL expression evaluates to that user name?

A. ${userName}
B. ${cookie.userName}
C. ${cookie.user.name}
D. ${cookies.userName[0]}
E. ${cookies.userName}[1]
F. ${cookies.get('userName')}
Answer: B

Q10. Given an EL function declared with:
11. <function>

12. <name>spin</name>
13. <function-class>com.example.Spinner</function-class>
14. <function-signature>
15. java.lang.String spinIt()
16. </function-signature>
17. </function>

Which two are true? (Choose two.)

A. The function method must have the signature:
public String spin().
B. The method must be mapped to the logical name "spin" in the web.xml file.
C. The function method must have the signature:
public String spinIt().
D. The function method must have the signature
public static String spin().
E. The function method must have the signature:
public static String spinIt().
F. The function class must be named Spinner, and must be in the package com.example.

Answer: E, F

Q11. You need to create a JSP that generates some JavaScript code to populate an array of strings used on the client-side. Which JSP code snippet will create this array?

A. MY_ARRAY = new Array();
<% for (int i = 0; i < serverArray.length; i++) {
MY_ARRAY[<%= i %>] = '<%= serverArray[i] %>';
} %>
B. MY_ARRAY = new Array();
<% for (int i = 0; i < serverArray.length; i++) {
MY_ARRAY[${i}] = '${serverArray[i]}';
} %>
C. MY_ARRAY = new Array();
<% for (int i = 0; i < serverArray.length; i++) { %>
MY_ARRAY[<%= i %>] = '<%= serverArray[i] %>';
<% } %>
D. MY_ARRAY = new Array();
<% for (int i = 0; i < serverArray.length; i++) { %>
MY_ARRAY[${i}] = '${serverArray[i]}';
<% } %>

Answer: C

Q12. Given:

6. <% int[] nums = {42, 420, 4200};
7. request.setAttribute("foo", nums); %>
Which two successfully translate and result in a value of true?

(Choose two.)
A. ${true or false}
B. ${requestScope[foo][0] > 500}
C. ${requestScope['foo'][1] = 420}
D. ${(requestScope['foo'][0] lt 50) && (3 gt 2)}
Answer: A, D

Q13.Click the Exhibit button.

Given:

11. <% com.example.Advisor advisor = new com.example.Advisor(); %>
12. <% request.setAttribute("foo", advisor); %>
Assuming there are no other "foo" attributes in the web application, which three are valid EL expressions for retrieving the advice property of advisor? (Choose three.)

```
 1. package com.example;
 2.
 3. public class Advisor {
 4.     private String advice="take out the
garbage";
 5.     public String getAdvice() {
 6.        return advice;
 7.     }
 8.     public void setAdvice(String advice) {
 9.        this.advice = advice;
10.     }
11. }
```

A. ${foo.advice}
B. ${request.foo.advice}
C. ${requestScope.foo.advice}
D. ${requestScope[foo[advice]]}
E. ${requestScope["foo"]["advice"]}
F. ${requestScope["foo"["advice"]]}

Answer: A, C, E

Q14. You are creating an error page that provides a user-friendly screen whenever a server exception occurs. You want to hide the stack trace, but you do want to provide the exception's error message to the user so the user can provide it to the customer service agent at your company. Which EL code snippet inserts this error message into the error page?
A. Message: ${exception.message}
B. Message: ${exception.errorMessage}
C. Message: ${request.exception.message}
D. Message: ${pageContext.exception.message}
E. Message: ${request.exception.errorMessage}
F. Message: ${pageContext.exception.errorMessage}
Answer: D

Q15. You are building a dating web site. The client's date of birth is collected along with lots of other information. You have created an EL function with the signature: calcAge(java.util.Date):int and it is assigned to the name, age, in the namespace, funct. In one of your JSPs you need to print a special message to clients who are younger than 25. Which EL code snippet will return true for this condition?

A. ${calcAge(client.birthDate) < 25}
B. ${calcAge[client.birthDate] < 25}
C. ${funct:age(client.birthDate) < 25}
D. ${funct:age[client.birthDate] < 25}
E. ${funct:calcAge(client.birthDate) < 25}
F. ${funct:calcAge[client.birthDate] < 25}

Answer: C

Q16. Given:

```
11. <% java.util.Map map = new java.util.HashMap();
12. request.setAttribute("map", map);
13. map.put("a", "b");
14. map.put("b", "c");
15. map.put("c", "d"); %>
16. <%-- insert code here --%>
```

Which three EL expressions, inserted at line 16, are valid and evaluate to "d"? (Choose three.)

A. ${map.c}
B. ${map[c]}
C. ${map["c"]}
D. ${map.map.b}
E. ${map[map.b]}
F. ${map.(map.b)}

Answer: A, C, E

-

Q17. Assume the tag handler for a st:simple tag extends SimpleTagSupport. In what way can scriptlet code be used in the body of st:simple?

A. set the body content type to JSP in the TLD
B. Scriptlet code is NOT legal in the body of st:simple.
C. add scripting-enabled="true" to the start tag for the st:simple element
D. add a pass-through Classic tag with a body content type of JSP to the body of st:simple, and place the scriptlet code in the body of that tag

Answer: B

Q18. You have built your own light-weight templating mechanism. Your servlets, which handle each request, dispatch the request to one of a small set of template JSP pages. Each template JSP controls the layout of the view by inserting the header, body, and footer elements into specific locations within the template page. The URLs for these three elements are stored in request-scoped variables called, headerURL, bodyURL, and footerURL, respectively. These attribute names are never used for other purposes. Which JSP code snippet should be used in the template JSP to insert the JSP content for the body of the page?

A. <jsp:insert page='${bodyURL}' />
B. <jsp:insert file='${bodyURL}' />
C. <jsp:include page='${bodyURL}' />
D. <jsp:include file='${bodyURL}' />
E. <jsp:insert page='<%= bodyURL %>' />
F. <jsp:include page='<%= bodyURL %>' />
Answer: C

Q19. Given tutorial.jsp:
2. <h1>EL Tutorial</h1>
3. <h2>Example 1</h2>
4.
5. Dear ${my:nickname(user)}
6.

Which, when added to the web application deployment descriptor, ensures that line 5 is included verbatim in the JSP output?
A. <jsp-config>
<url-pattern>*.jsp</url-pattern>
<el-ignored>true</el-ignored>
</jsp-config>
B. <jsp-config>
<url-pattern>*.jsp</url-pattern>
<isELIgnored>true</isELIgnored>
</jsp-config>
C. <jsp-config>
<jsp-property-group>
<el-ignored>*.jsp</el-ignored>
</jsp-property-group>
</jsp-config>
D. <jsp-config>
<jsp-property-group>
<url-pattern>*.jsp</url-pattern>
<el-ignored>true</el-ignored>
</jsp-property-group>
</jsp-config>
E. <jsp-config>
<jsp-property-group>
<url-pattern>*.jsp</url-pattern>
<isElIgnored>true</isElIgnored>
</jsp-property-group>
</jsp-config>

Answer: D

Q 20. All of your JSPs need to have a link that permits users to email the web master. This web application is licensed to many small businesses, each of which have a different email address for the web master. You have decided to use a context parameter that you specify in the deployment descriptor, like this:

42. <context-param>
43. <param-name>webmasterEmail</param-name>
44. <param-value>master@example.com</param-value>
45. </context-param>

Which JSP code snippet creates this email link?

A. contact us
B. contact us
C. contact us
D. contact us

Answer: D

Q21. A web application allows the HTML title banner to be set using a servlet context initialization parameter called titleStr. Which two properly set the title in this scenario? (Choose two.)

A. <title>${titleStr}</title>
B. <title>${initParam.titleStr}</title>
C. <title>${params[0].titleStr}</title>
D. <title>${paramValues.titleStr}</title>
E. <title>${initParam['titleStr']}</title>
F. <title>${servletParams.titleStr}</title>
G. <title>${request.get("titleStr")}</title>

Answer: B, E

Q22. You are building a dating service web site. Part of the form to submit a client's profile is a group of radio buttons for the person's hobbies:

20. <input type='radio' name='hobbyEnum' value='HIKING'>Hiking
21. <input type='radio' name='hobbyEnum' value='SKIING'>Skiing
22. <input type='radio' name='hobbyEnum' value='SCUBA'>SCUBA Diving
23. <!-- and more options -->

After the user submits this form, a confirmation screen is displayed with these hobbies listed. Assume
that an application-scoped variable, hobbies, holds a map between the Hobby enumerated type and the
display name.

Which EL code snippet will display Nth element of the user's selected hobbies?

A. ${hobbies[hobbyEnum[N]}
B. ${hobbies[paramValues.hobbyEnum[N]]}
C. ${hobbies[paramValues@'hobbyEnum'@N]}
D. ${hobbies.get(paramValues.hobbyEnum[N])}
E. ${hobbies[paramValues.hobbyEnum.get(N)]}

Answer: B

Q23. Given a web application in which the request parameter productID contains a product identifier. Which two EL expressions evaluate the value of the productID? (Choose two.)

A. ${productID}
B. ${param.productID}
C. ${params.productID}
D. ${params.productID[1]}
E. ${paramValues.productID}
F. ${paramValues.productID[0]}
G. ${pageContext.request.productID}

Answer: B, F

Q 24. You are building a web application with a scheduling component. On the JSP, you need to show the current date, the date of the previous week, and the date of the next week. To help you present this information, you have created the following EL functions in the 'd' namespace:
name: curDate; signature: java.util.Date currentDate()
name: addWeek; signature: java.util.Date addWeek(java.util.Date, int)
name: dateString; signature: java.util.String getDateString(java.util.Date)
Which EL code snippet will generate the string for the previous week?
A. ${d:dateString(addWeek(curDate(), -1))}
B. ${d:dateString[addWeek[curDate[], -1]]}
C. ${d:dateString[d:addWeek[d:curDate[], -1]]}
D. ${d:dateString(d:addWeek(d:curDate(), -1))}

Answer: D

Q25. You are building a dating web site. The client's date of birth is collected along with lots of other information. The Person class has a derived method, getAge():int, which returns the person's age calculated from the date of birth and today's date. In one of your JSPs you need to print a special message to clients within the age group of 25 through 35. Which two EL code snippets will return true for this condition?
(Choose two.)

A. ${client.age in [25,35]}
B. ${client.age between [25,35]}
C. ${client.age between 25 and 35}
D. ${client.age <= 35 && client.age >= 25}
E. ${client.age le 35 and client.age ge 25}
F. ${not client.age > 35 && client.age < 25}

Answer: D, E

Unit 4 & 5: Building JSP Pages Using Tag Libraries and Custom Tag Library

- For a custom tag library or a library of Tag Files, create the 'taglib' directive for a JSP page.
- Given a design goal, create the custom tag structure in a JSP page to support that goal.
- Given a design goal, use an appropriate JSP Standard Tag Library (JSTL v1.1) tag from the "core" tag library.
- Describe the semantics of the "Classic" custom tag event model when each event method (doStartTag, doAfterBody, and doEndTag) is executed, and explain what the return value for each event method means; and write a tag handler class.
- Using the PageContext API, write tag handler code to access the JSP implicit variables and access web application attributes.
- Given a scenario, write tag handler code to access the parent tag and an arbitrary tag ancestor.
- Describe the semantics of the "Simple" custom tag event model when the event method (doTag) is executed; write a tag handler class; and explain the constraints on the JSP content within the tag.
- Describe the semantics of the Tag File model; describe the web application structure for tag files; write a tag file; and explain the constraints on the JSP content in the body of the tag.

Q1. You have built a collection of custom tags for your web application. The TLD file is located in the file: /WEB-INF/myTags.xml. You refer to these tags in your JSPs using the symbolic name: myTags. Which deployment descriptor element must you use to make this link between the symbolic name and the TLD file name?
A. <taglib>
<name>myTags</name>
<location>/WEB-INF/myTags.xml</location>
</taglib>
B. <tags>
<name>myTags</name>
<location>/WEB-INF/myTags.xml</location>
</tags>
C. <tags>
<tags-uri>myTags</taglib-uri>
<tags-location>/WEB-INF/myTags.xml</tags-location>
</tags>
D. <taglib>
<taglib-uri>myTags</taglib-uri>
<taglib-location>/WEB-INF/myTags.xml</taglib-location>
</taglib>
Answer: D

Q2. Click the Exhibit button.
The attribute "name" has a value of "Foo,"
What is the result if this tag handler's tag is invoked?

```
 5. public class MyTagHandler extends
TagSupport {
 6.   public int doStartTag() throws
JspException {
 7.       try {
 8.         Writer out =
pageContext.getResponse().getWriter();
 9.         String name =
pageContext.findAttribute("name");
10.         out.print(name);
11.       } catch(Exception ex) { /* handle
exception */ }
12.       return SKIP_BODY;
13.   }
14
15.   public int doAfterBody() throws
JspException {
16.       try {
17.         Writer out =
pageContext.getResponse().getWriter();
18.         out.print("done");
19.       } catch(Exception ex) { /* handle
exception */ }
20.       return EVAL_PAGE;
21.   }
...
42. }
```

A. Foo
B. done
C. Foodone
D. An exception is thrown at runtime.
E. No output is produced from this code.
F. Compilation fails because of an error in this code.
Answer: A

Q3. You are building a web application that will be used throughout the European Union; therefore, it has significant internationalization requirements. You have been tasked to create a custom tag that generates a message using the java.text.MessageFormat class. The tag will take the resourceKey attribute and a variable number of argument attributes with the format, arg<N>. Here is an example use of this tag and its output:
<t:message resourceKey='diskFileMsg' arg0='MyDisk' arg1='1247' />
generates:The disk "MyDisk" contains 1247 file(s).
Which Simple tag class definition accomplishes this goal of handling a variable number of tag attributes?

A. public class MessageTag extends SimpleTagSupport
implements VariableAttributes {
private Map attributes = new HashMap();

```
public void setVariableAttribute(String uri,
String name, Object value) {
this.attributes.put(name, value);
}
// more tag handler methods
}
```

B. The Simple tag model does NOT support a variable number of attributes.

```
C. public class MessageTag extends SimpleTagSupport
implements DynamicAttributes {
private Map attributes = new HashMap();
public void putAttribute(String name, Object value) {
this.attributes.put(name, value);
}
// more tag handler methods
}
D. public class MessageTag extends SimpleTagSupport
implements VariableAttributes {
private Map attributes = new HashMap();
public void putAttribute(String name, Object value) {
this.attributes.put(name, value);
}
// more tag handler methods
}
E. public class MessageTag extends SimpleTagSupport
implements DynamicAttributes {
private Map attributes = new HashMap();
public void setDynamicAttribute(String uri, String name,
Object value) {
this.attributes.put(name, value);
}
// more tag handler methods
}
```

Q4. Given the JSP code:

```
<% request.setAttribute("foo", "bar"); %>
```

and the Classic tag handler code:

```
5. public int doStartTag() throws JspException {
6. // insert code here
7. // return int
8. }
```

Assume there are no other "foo" attributes in the web application.
Which invocation on the pageContext object, inserted at line 6, assigns "bar" to the variable x?

```
A. String x = (String) pageContext.getAttribute("foo");
B. String x = (String) pageContext.getRequestScope("foo");
C. It is NOT possible to access the pageContext object from within doStartTag.
D. String x = (String)
pageContext.getRequest().getAttribute("foo");
E. String x = (String) pageContext.getAttribute("foo",
PageContext.ANY_SCOPE);
```

Answer: D

Q5. Which two statements about tag files are true? (Choose two.)

A. Classic tag handlers and tag files CANNOT reside in the same tag library.
B. A file named foo.tag, located in /WEB-INF/tags/bar, is recognized as a tag file by the container.
C. A file named foo.tag, bundled in a JAR file but NOT defined in a TLD, triggers a container translation error.
D. A file named foo.tag, located in a web application's root directory, is recognized as a tag file by the container.
E. If files foo1.tag and foo2.tag both reside in /WEB-INF/tags/bar, the container will consider them part of the same tag library.

Answer: B, E

Q6. The sl:shoppingList and sl:item tags output a shopping list to the response and are used as follows:
11. <sl:shoppingList>
12. <sl:item name="Bread" />
13. <sl:item name="Milk" />
14. <sl:item name="Eggs" />
15. </sl:shoppingList>

The tag handler for sl:shoppingList is ShoppingListTag and the tag handler for sl:item is ItemSimpleTag.
ShoppingListTag extends BodyTagSupport and ItemSimpleTag extends SimpleTagSupport.
Which is true?

A. ItemSimpleTag can find the enclosing instance of ShoppingListTag by calling getParent() and casting the
result to ShoppingListTag.
B. ShoppingListTag can find the child instances of ItemSimpleTag by calling super.getChildren() and casting
each to an ItemSimpleTag.
C. It is impossible for ItemSimpleTag and ShoppingListTag to find each other in a tag hierarchy because one is
a Simple tag and the other is a Classic tag.
D. ShoppingListTag can find the child instances of ItemSimpleTag by calling getChildren() on the PageContext
and casting each to an ItemSimpleTag.
E. ItemSimpleTag can find the enclosing instance of ShoppingListTag by calling findAncestorWithClass() on
the PageContext and casting the result to ShoppingListTag.

Answer: A

Q7. Assume that a news tag library contains the tags lookup and item:
lookup Retrieves the latest news headlines and executes the tag body once for each headline.

Exposes a NESTED page-scoped attribute called headline of type com.example.Headline containing details for that headline.
item Outputs the HTML for a single news headline. Accepts an attribute info of type com.example.Headline containing details for the headline to be rendered.Which snippet of JSP code returns the latest news headlines in an HTML table, one per row?

A. <table>
<tr>
<td>
<news:lookup />
<news:item info="${headline}" />
</td>
</tr>
</table>

B. <news:lookup />
<table>
<tr>
<td><news:item info="${headline}" /></td>
</tr>
</table>

C. <table>
<news:lookup>
<tr>
<td><news:item info="${headline}" /></td>
</tr>
</news:lookup>
</table>

D. <table>
<tr>
<news:lookup>
<td><news:item info="${headline}" /></td>
</news:lookup>
</tr>
</table>

Answer: C

Q8. Which JSTL code snippet can be used to perform URL rewriting?

A. <a href='<c:url url="foo.jsp"/>' />
B. <a href='<c:link url="foo.jsp"/>' />
C. <a href='<c:url value="foo.jsp"/>' />
D. <a href='<c:link value="foo.jsp"/>' />

Answer: C

Q9. Assume the scoped attribute priority does NOT yet exist. Which two create and set a new request-scoped attribute priority to the value "medium"? (Choose two.)

A. ${priority = 'medium'}
B. ${requestScope['priority'] = 'medium'}
C. <c:set var="priority" value="medium" />
D. <c:set var="priority" scope="request">medium</c:set>
E. <c:set var="priority" value="medium" scope="request" />
F. <c:set property="priority" scope="request">medium</c:set>
G. <c:set property="priority" value="medium" scope="request" />

Answer: D, E

Q10. You are creating a JSP page to display a collection of data. This data can be displayed in several different ways so the architect on your project decided to create a generic servlet that generates a comma-delimited string so that various pages can render the data in different ways. This servlet takes on request parameter: objectID. Assume that this servlet is mapped to the URL pattern:/WEB-INF/data.

In the JSP you are creating, you need to split this string into its elements separated by commas and generate an HTML <ul> list from the data.

Which JSTL code snippet will accomplish this goal?

A. <c:import varReader='dataString' url='/WEB-INF/data'>
<c:param name='objectID' value='${currentOID}' />
</c:import>

<c:forTokens items'${dataString.split(",")}' var='item'>
${item}
</c:forTokens>

B. <c:import varReader='dataString' url='/WEB-INF/data'>
<c:param name='objectID' value='${currentOID}' />
</c:import>

<c:forTokens items'${dataString}' delims=',' var='item'>
${item}
</c:forTokens>

C. <c:import var='dataString' url='/WEB-INF/data'>
<c:param name='objectID' value='${currentOID}' />
</c:import>

<c:forTokens items'${dataString.split(",")}' var='item'>
${item}
</c:forTokens>

D. <c:import var='dataString' url='/WEB-INF/data'>
<c:param name='objectID' value='${currentOID}' />
</c:import>


```
<c:forTokens items'${dataString}' delims=',' var='item'>
<li>${item}</li>
</c:forTokens>
</ul>
```

Answer: D

Q11. Which three are true about TLD files? (Choose three.)

A. The web container recognizes TLD files placed in any subdirectory of WEB-INF.
B. When deployed inside a JAR file, TLD files must be in the META-INF directory, or a subdirectory of it.
C. A tag handler's attribute must be included in the TLD file only if the attribute can accept request-time expressions.
D. The web container can generate an implicit TLD file for a tag library comprised of both simple tag handlers and tag files.
E. The web container can automatically extend the tag library map described in a web.xml file by including entries extracted from the web application's TLD files.

Answer: A, B, E

Q12. Your management has required that all JSPs be created to generate XHTML-compliant content and to facilitate that decision, you are required to create all JSPs using the JSP Document format. In the reviewOrder.jspx page, you need to use several core JSTL tags to process the collection of order items in the customer's shopping cart. Which JSP code snippets must you use in the reviewOrder.jspx page?

A.
```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
version="2.0">
<jsp:directive.taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" />
<!-- page content -->
</html>
```
B.
```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
version="2.0"
xmlns:c="http://java.sun.com/jsp/jstl/core">
<!-- page content -->
</html>
```
C.
```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
version="2.0">
<jsp:directive.taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" />
<!-- page content -->
</jsp:root>
```
D.
```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
version="2.0"
xmlns:c="http://java.sun.com/jsp/jstl/core">
<!-- page content -->
</jsp:root>
```
Answer: D

Q13. Which two JSTL URL-related tags perform URL rewriting? (Choose two.)

A. url **B. link**
C. param **D. import**
E. redirect

Answer: A, E

Q14. A custom JSP tag must be able to support an arbitrary number of attributes whose names are unknown when the tag class is designed. Which two are true? (Choose two.)
A. The <body-content> element in the echo tag TLD must have the value JSP.
B. The echo tag handler must define the setAttribute(String key, String value) method.
C. The <dynamic-attributes>true</dynamic-attributes> element must appear in the echo tag TLD.
D. The class implementing the echo tag handler must implement the javax.servlet.jsp.tagext.IterationTag interface.
E. The class implementing the echo tag handler must implement the javax.servlet.jsp.tagext.DynamicAttributes interface.

Answer: C, E

Q15. After a merger with another small business, your company has inherited a legacy WAR file but the original source files were lost. After reading the documentation of that web application, you discover that the WAR file contains a useful tag library that you want to reuse in your own webapp packaged as a WAR file.
What do you need to do to reuse this tag library?

A. Simply rename the legacy WAR file as a JAR file and place it in your webapp's library directory.
B. Unpack the legacy WAR file, move the TLD file to the META-INF directory, repackage the whole thing as a JAR file, and place that JAR file in your webapp's library directory.
C. Unpack the legacy WAR file, move the TLD file to the META-INF directory, move the class files to the top-level directory, repackage the whole thing as a JAR file, and place that JAR file in your webapp's library directory.
D. Unpack the legacy WAR file, move the TLD file to the META-INF directory, move the class files to the top-level directory, repackage the WAR, and place that WAR file in your webapp's WEB-INF directory.
Answer: C

Q:16. You want to create a valid directory structure for your Java EE web application, and your application uses tag files and a JAR file. Which three must be located directly in your WEB-INF directory (NOT in a subdirectory of WEB-INF)? (Choose three.)

A. The JAR file
B. A directory called lib
C. A directory called tags
D. A directory called TLDs
E. A directory called classes

F. A directory called META-INF

Answer: B, C, E

Q17. Assume the custom tag my:errorProne always throws a java.lang.Runtime Exception with the message "File not found."
An error page has been configured for this JSP page.
Which option prevents the exception thrown by my:errorProne from invoking the error page mechanism, and outputs the message "File not found" in the response?

A. <c:try catch="ex">
<my:errorProne />
</c:try>
${ex.message}
B. <c:catch var="ex">
<my:errorProne />
</c:catch>
${ex.message}
C. <c:try>
<my:errorProne />
</c:try>
<c:catch var="ex" />
${ex.message}
D. <c:try>
<my:errorProne />
<c:catch var="ex" />
${ex.message}
</c:try>
E. <my:errorProne>
<c:catch var="ex">
${ex.message}
</c:catch>
</my:errorProne>
Answer: B

Q18. A JSP page contains a taglib directive whose uri attribute has the value dbtags. Which XML element within the web application deployment descriptor defines the associated TLD?

A. <tld>
<uri>dbtags</uri>
<location>/WEB-INF/tlds/dbtags.tld</location>
</tld>
B. <taglib>
<uri>dbtags</uri>
<location>/WEB-INF/tlds/dbtags.tld</location>
</taglib>
C. <tld>
<tld-uri>dbtags</tld-uri>
<tld-location>/WEB-INF/tlds/dbtags.tld</tld-location>

</tld>
D. <taglib>
<taglib-uri>dbtags</taglib-uri>
<taglib-location>
/WEB-INF/tlds/dbtags.tld
</taglib-location>
</taglib>

Answer: D

Q19. Click the Exhibit button.

Assuming the tag library in the exhibit is imported with the prefix stock, which custom tag invocation outputs the contents of the variable exposed by the quote tag?

```
 1. <?xml version="1.0" encoding="UTF-8" ?>
 2.
 3. <taglib
xmlns="http://java.sun.com/xml/ns/j2ee"
 4.    xmlns:xsi="http://www.w3.org/2001/XMLSchem
a-instance"
 5.    xsi:schemaLocation="http://java.sun.com/xm
l/ns/j2ee web-jsptaglibrary_2_0.xsd"
 6.    version="2.0">
 7.    <tlib-version>1.0</tlib-version>
 8.    <short-name>stock</short-name>
 9.    <uri>http://example.com/tld/stock</uri>
10.    <tag>
11.      <name>quote</name>
12.
<tag-class>com.example.QuoteTag</tag-class>
13.      <body-content>empty</body-content>
14.      <variable>
15.        <name-from-attribute>var</name-from-att
ute>
16.        <scope>AT_BEGIN</scope>
17.      </variable>
18.      <attribute>
19.        <name>symbol</name>
20.        <required>true</required>
21.        <rtexprvalue>true</rtexprvalue>
22.      </attribute>
23.      <attribute>
24.        <name>var</name>
25.        <required>true</required>
26.        <rtexprvalue>false</rtexprvalue>
27.      </attribute>
28.    </tag>
29. </taglib>
```

A. <stock:quote symbol="SUNW" />
${var}
B. ${var}
<stock:quote symbol="SUNW" />

C. <stock:quote symbol="SUNW">
${var}
</stock:quote>
D. <stock:quote symbol="SUNW" var="quote" />
${quote}
E. <stock:quote symbol="SUNW" var="quote">
<%= quote %>
</stock:quote>

Answer: D

Q20. You are creating a JSP page to display a collection of data. This data can be displayed in several different ways so the architect on your project decided to create a generic servlet that generates a comma-delimited string so that various pages can render the data in different ways. This servlet takes on request parameter: objectID. Assume that this servlet is mapped to the URL pattern: /WEB-INF/data.

In the JSP you are creating, you need to split this string into its elements separated by commas and generate an HTML <ul> list from the data.
Which JSTL code snippet will accomplish this goal?

A. <c:import varReader='dataString' url='/WEB-INF/data'>
<c:param name='objectID' value='${currentOID}' />
</c:import>

<c:forTokens items'${dataString.split(",")}' var='item'>
${item}
</c:forTokens>

B. <c:import varReader='dataString' url='/WEB-INF/data'>
<c:param name='objectID' value='${currentOID}' />
</c:import>

<c:forTokens items'${dataString}' delims=',' var='item'>
${item}
</c:forTokens>

C. <c:import var='dataString' url='/WEB-INF/data'>
<c:param name='objectID' value='${currentOID}' />
</c:import>

<c:forTokens items'${dataString.split(",")}' var='item'>
${item}
</c:forTokens>

D. <c:import var='dataString' url='/WEB-INF/data'>
<c:param name='objectID' value='${currentOID}' />
</c:import>

<c:forTokens items'${dataString}' delims=',' var='item'>
${item}
</c:forTokens>

Answer: D

Q21. A web application contains a tag file called beta.tag in /WEB-INF/tags/alpha.
A JSP page called sort.jsp exists in the web application and contains only this JSP code:
1. <%@ taglib prefix="x"
2. tagdir="/WEB-INF/tags/alpha" %>
3. <x:beta />
The sort.jsp page is requested.
Which two are true? (Choose two.)

A. Tag files can only be accessed using a tagdir attribute.
B. The sort.jsp page translates successfully and invokes the tag defined by beta.tag.
C. The sort.jsp page produces a translation error because a taglib directive must always have a uri attribute.
D. Tag files can only be placed in /WEB-INF/tags, and NOT in any subdirectories of /WEB-INF/tags.
E. The tagdir attribute in line 2 can be replaced by a uri attribute if a TLD referring to beta.tag is created and added to the web application.
F. The sort.jsp page produces a translation error because the tagdir attribute on lines 1-2 specifies a directory other than /WEB-INF/tags, which is illegal.

Answer: B, E

Q22. Given a JSP page:

11. <n:recurse>
12. <n:recurse>
13. <n:recurse>
14. <n:recurse />
15. </n:recurse>
16. </n:recurse>
17. </n:recurse>

The tag handler for n:recurse extends SimpleTagSupport.
Assuming an n:recurse tag can either contain an empty body or another n:recurse tag, which strategy allows the tag handler for n:recurse to output the nesting depth of the deepest n:recurse tag?
A. It is impossible to determine the deepest nesting depth because it is impossible for tag handlers that extend SimpleTagSupport to communicate with their parent and child tags.
B. Create a private non-static attribute in the tag handler class called count of type int initialized to 0. Increment count in the doTag method. If the tag has a body, invoke the fragment for that body. Otherwise, output the value of count.
C. Start a counter at 1. Call getChildTags(). If it returns null, output the value of the counter. Otherwise, increment counter and continue from where getChildTags() is called. Skip processing of the body.

D. If the tag has a body, invoke the fragment for that body.Otherwise, start a counter at 1. Call getParent(). If it returns null, output the value of the counter Otherwise, increment the counter and continue from where getParent() is called.
Answer: D

Q23. Click the Exhibit button.
The h:highlight tag renders its body, highlighting an arbitrary number of words, each of which is passed as an attribute (word1, word2, ...). For example, a JSP page can invoke the h:highlight tag as follows:
11. <h:highlight color="yellow" word1="high" word2="low">
12. high medium low
13. </h:highlight>
Given that HighlightTag extends SimpleTagSupport, which three steps are necessary to implement the tag handler for the highlight tag? (Choose three).

```
 1. <?xml version="1.0" encoding="UTF-8" ?>
 2.
 3. <taglib
xmlns="http://java.sun.com/xml/ns/j2ee"
 4.    xmlns:xsi="http://www.w3.org/2001/XMLSchem
a-instance"
 5.    xsi:schemaLocation="http://java.sun.com/xm
l/ns/j2ee web-jsptaglibrary_2_0.xsd"
 6.    version="2.0">
 7.    <tlib-version>1.0</tlib-version>
 8.    <short-name>h</short-name>
 9.
<uri>http://example.com/tld/highlight</uri>
10.    <tag>
11.      <name>highlight</name>
12.      <tag-class>com.example.HighlightTag</tag-
lass>
13.
<body-content>scriptless</body-content>
14.      <attribute>
15.        <name>color</name>
16.        <required>true</required>
17.      </attribute>
18.
<dynamic-attributes>true</dynamic-attributes>
19.    </tag>
20. </taglib>
```

A. add a doTag method
B. add a doStartTag method
C. add a getter and setter for the color attribute
D. create and implement a TagExtraInfo class
E. implement the DynamicAttributes interface
F. add a getter and setter for the word1 and word2 attributes
Answer: A, C, E

Q24. Given:

5. public class MyTagHandler extends TagSupport {
6. public int doStartTag() throws JspException {
7. try {
8. // insert code here

```
9. } catch(Exception ex) { /* handle exception */ }
10. return super.doStartTag();
11. }
...
42. }
```

Which code snippet, inserted at line 8, causes the value foo to be output?

```
A. JspWriter w = pageContext.getOut();
w.print("foo");
B. JspWriter w = pageContext.getWriter();
w.print("foo");
C. JspWriter w = new JspWriter(pageContext.getWriter());
w.print("foo");
D. JspWriter w = new JspWriter(pageContext.getResponse());
w.print("foo");
```

Answer: A

Q25. Given:

```
6. <myTag:foo bar='42'>
7. <%="processing" %>
8. </myTag:foo>
```

and a custom tag handler for foo which extends TagSupport.

Which two are true about the tag handler referenced by foo? (Choose two.)

A. The doStartTag method is called once.
B. The doAfterBody method is NOT called.
C. The EVAL_PAGE constant is a valid return value for the doEndTag method.
D. The SKIP_PAGE constant is a valid return value for the doStartTag method.
E. The EVAL_BODY_BUFFERED constant is a valid return value for the doStartTag method.

Answer: A, C

Q26. Which three are valid values for the body-content attribute of a tag directive in a tag file? (Choose three.)
A. EL
B. JSP
C. empty
D. dynamic
E. scriptless
F. tagdependent

Answer: C, E, F

Q27. You have a new IT manager that has mandated that all JSPs must be refactored to include no scritplet code. The IT manager has asked you to enforce this. Which deployment descriptor element will satisfy this constraint?

A. <jsp-property-group>
<url-pattern>*.jsp</url-pattern>
<permit-scripting>false</permit-scripting>
</jsp-property-group>
B. <jsp-config>
<url-pattern>*.jsp</url-pattern>
<permit-scripting>false</permit-scripting>
</jsp-config>
C. <jsp-config>
<url-pattern>*.jsp</url-pattern>
<scripting-invalid>true</scripting-invalid>
</jsp-config>
D. <jsp-property-group>
<url-pattern>*.jsp</url-pattern>
<scripting-invalid>true</scripting-invalid>
</jsp-property-group>

Answer: D

Q28. Assume the tag handler for a st:simple tag extends SimpleTagSupport. In what way can scriptlet code be used in the body of st:simple?

A. set the body content type to JSP in the TLD
B. Scriptlet code is NOT legal in the body of st:simple.
C. add scripting-enabled="true" to the start tag for the st:simple element
D. add a pass-through Classic tag with a body content type of JSP to the body of st:simple, and place the
scriptlet code in the body of that tag
Answer: B

Q29. Which statement is true if the doStartTag method returns EVAL_BODY_BUFFERED ?

A. The tag handler must implement BodyTag.
B. The doAfterBody method is NOT called.
C. The setBodyContent method is called once.
D. It is never legal to return EVAL_BODY_BUFFERED from doStartTag.

Answer: C

Q30. You are creating a library of custom tags that mimic the HTML form tags. When the user submits a form that fails validation, the JSP form is forwarded back to the user. The <t:textField> tag must support the ability to re-populate the form field with the request parameters from the user's last request. For example, if the user entered "Samantha" in the text field called firstName, then the form is re-populated like this:

<input type='text' name='firstName' value='Samantha' />

Which tag handler method will accomplish this goal?

A. public int doStartTag() throws JspException {
JspContext ctx = getJspContext();
String value = ctx.getParameter(this.name);
if (value == null) value = "";
JspWriter out = pageContext.getOut();
try {
out.write(String.format(INPUT, this.name, value));
} (Exception e) { throw new JspException(e); }
return SKIP_BODY;
}
private static String INPUT
= "<input type='text' name='%s' value='%s' />";
B. public void doTag() throws JspException {
JspContext ctx = getJspContext();
String value = ctx.getParameter(this.name);
if (value == null) value = "";
JspWriter out = pageContext.getOut();
try {
out.write(String.format(INPUT, this.name, value));
} (Exception e) { throw new JspException(e); }
}
private static String INPUT
= "<input type='text' name='%s' value='%s' />";
C. public int doStartTag() throws JspException {
ServletRequet request = pageContext.getRequest();
String value = request.getParameter(this.name);
if (value == null) value = "";
JspWriter out = pageContext.getOut();
try {
out.write(String.format(INPUT, this.name, value));
} (Exception e) { throw new JspException(e); }
return SKIP_BODY;
}
private static String INPUT
= "<input type='text' name='%s' value='%s' />";
D. public void doTag() throws JspException {
ServletRequet request = pageContext.getRequest();
String value = request.getParameter(this.name);
if (value == null) value = "";
JspWriter out = pageContext.getOut();
try {
out.write(String.format(INPUT, this.name, value));
} (Exception e) { throw new JspException(e); }
}
private static String INPUT

= "<input type='text' name='%s' value='%s' />";

Answer: C

Q31. Which two directives are applicable only to tag files? (Choose two.)

A. tag
B. page
C. taglib
D. include
E. variable
Answer: A, E

Q32. The tl:taskList and tl:task tags output a set of tasks to the response and are used as follows:

11. <tl:taskList>
12. <tl:task name="Mow the lawn" />
13. <tl:task name="Feed the dog" />
14. <tl:task name="Do the laundry" />
15. </tl:taskList>
The tl:task tag supplies information about a single task while the tl:taskList tag does the final output. The tag handler for tl:taskList is TaskListTag. The tag handler for tl:task is TaskTag. Both tag handlers
extend BodyTagSupport.
Which allows the tl:taskList tag to get the task names from its nested tl:task children?

A. It is impossible for a tag handler that extends BodyTagSupport to communicate with its parent and child
tags.
B. In the TaskListTag.doStartTag method, call super.getChildTags() and iterate through the results. Cast each
result to a TaskTag and call getName().
C. In the TaskListTag.doStartTag method, call getChildTags() on the PageContext and iterate through the
results. Cast each result to a TaskTag and call getName().
D. Create an addTaskName method in TaskListTag. Have the TaskListTag.doStartTag method, return
BodyTag.EVAL_BODY_BUFFERED. In the TaskTag.doStartTag method, call super.getParent(), cast it to a
TaskListTag, and call addTaskName().
E. Create an addTaskName method in TaskListTag. Have the TaskListTag.doStartTag method, return
BodyTag.EVAL_BODY_BUFFERED. In the TaskTag.doStartTag method, call findAncestorWithClass() on the
PageContext, passing TaskListTag as the class to find. Cast the result to TaskListTag and call addTaskName().

Answer: D

Q33. You are developing several tag libraries that will be sold for development of third-party web applications. You are about to publish the first three libraries as JAR files:container-tags.jar, advanced-html-form-tags.jar, and basic-html-form-tags.jar. Which two techniques are appropriate for packaging the TLD files for these tag libraries? (Choose two.)

A. The TLD must be located within the WEB-INF directory of the JAR file.
B. The TLD must be located within the META-INF directory of the JAR file.
C. The TLD must be located within the META-INF/tld/ directory of the JAR file.
D. The TLD must be located within a subdirectory of WEB-INF directory of the JAR file.
E. The TLD must be located within a subdirectory of META-INF directory of the JAR file.
F. The TLD must be located within a subdirectory of META-INF/tld/ directory of the JAR file.
Answer: B,

Q34. A custom tag is defined to take three attributes. Which two correctly invoke the tag within a JSP page? (Choose two.)

A. <prefix:myTag a="foo" b="bar" c="baz" />
B. <prefix:myTag attributes={"foo","bar","baz"} />
C. <prefix:myTag jsp:attribute a="foo" b="bar" c="baz" />
D. <prefix:myTag>
<jsp:attribute a:foo b:bar c:baz />
</prefix:myTag>
E. <prefix:myTag>
<jsp:attribute ${"foo", "bar", "baz"} />
</prefix:myTag>
F. <prefix:myTag>
<jsp:attribute a="foo" b="bar" c="baz"/>
</prefix:myTag>
G. <prefix:myTag>
<jsp:attribute name="a">foo</jsp:attribute>
<jsp:attribute name="b">bar</jsp:attribute>
<jsp:attribute name="c">baz</jsp:attribute>
</prefix:myTag>

Answer: A, G

Q35. You have been contracted to create a web site for a free dating service. One feature is the ability for one client to send a message to another client, which is displayed in the latter client's private page. Your contract explicitly states that security is a high priority. Therefore, you need to prevent cross-site hacking in which one user inserts JavaScript code that is then rendered and invoked when another user views that content. Which two JSTL code snippets will prevent cross-site hacking in the scenario above? (Choose two.)

A. <c:out>${message}</c:out>
B. <c:out value='${message}' />
C. <c:out value='${message}' escapeXml='true' />
D. <c:out eliminateXml='true'>${message}</c:out>
E. <c:out value='${message}' eliminateXml='true' />

Answer: B, C

Q36. Click the Exhibit button.
Assuming the tag library in the exhibit is imported with the prefix forum, which custom tag invocation produces a translation error in a JSP page?

```
 1. <?xml version="1.0" encoding="UTF-8" ?>
 2.
 3. <taglib
xmlns="http://java.sun.com/xml/ns/j2ee"
 4.    xmlns:xsi="http://www.w3.org/2001/XMLSchem
a-instance"
 5.    xsi:schemaLocation="http://java.sun.com/xm
l/ns/j2ee web-jsptaglibrary_2_0.xsd"
 6.    version="2.0">
 7.    <tlib-version>1.0</tlib-version>
 8.    <short-name>forum</short-name>
 9.    <uri>http://example.com/tld/forum</uri>
10.    <tag>
11.      <name>message</name>
12.
<tag-class>com.example.MessageTag</tag-class>
13.
<body-content>scriptless</body-content>
14.      <attribute>
15.        <name>from</name>
16.        <rtexprvalue>true</rtexprvalue>
17.      </attribute>
18.      <attribute>
19.        <name>subject</name>
20.        <required>false</required>
21.        <rtexprvalue>true</rtexprvalue>
22.      </attribute>
23.    </tag>
24. </taglib>
```

A. <forum:message from="My Name" subject="My Subject" />
B. <forum:message subject="My Subject">
My message body.
</forum:message>
C. <forum:message from="My Name" subject="${param.subject}">
${param.body}
</forum:message>
D. <forum:message from="My Name" subject="My Subject">
<%= request.getParameter("body") %>
</forum:message>
E. <forum:message from="My Name"
subject="<%= request.getParameter("subject") %>">
My message body.
</forum:message>
Answer: D

Q37. Which JSTL code snippet can be used to import content from another web resource?

A. <c:import url="foo.jsp"/>
B. <c:import page="foo.jsp"/>
C. <c:include url="foo.jsp"/>
D. <c:include page="foo.jsp"/>
E. Importing cannot be done in JSTL. A standard action must be used instead.

Answer: A

Q38. Click the Exhibit button.
Assume the tag library in the exhibit is placed in a web application in the path /WEB-INF/tld/example.tld.
1.
2. <ex:hello />
Which JSP code, inserted at line 1, completes the JSP code to invoke the hello tag?

A. <%@ taglib prefix="ex" uri="/WEB-INF/tld" %>
B. <%@ taglib uri="/WEB-INF/tld/example.tld" %>
C. <%@ taglib prefix="ex"
uri="http://localhost:8080/tld/example.tld" %>
D. <%@ taglib prefix="ex"
uri="http://example.com/tld/example" %>

Answer: D

Q39. Which JSTL code snippet produces the output "big number" when X is greater than 42, but outputs "small number" in all other cases?

A. <c:if test='<%= (X > 42) %>'>
<c:then>big number</c:then>
<c:else>small number</c:else>
</c:if>
B. <c:if>
<c:then test='<%= (X > 42) %>'>big number</c:then>
<c:else>small number</c:else>
</c:if>
C. <c:choose test='<%= (X > 42) %>'>
<c:then>big number</c:when>
<c:else>small number</c:otherwise>
</c:choose>
D. <c:choose test='<%= (X > 42) %>'>
<c:when>big number</c:when>
<c:otherwise>small number</c:otherwise>
</c:choose>
E. <c:choose>
<c:when test='<%= (X > 42) %>'>big number</c:when>
<c:otherwise>small number</c:otherwise>

</c:choose>

Answer: E

Q40. After a merger with another small business, your company has inherited a legacy WAR file but the original source files were lost. After reading the documentation of that web application, you discover that the WAR file contains a useful tag library that you want to reuse in your own webapp packaged as a WAR file.

What do you need to do to reuse this tag library?

A. Simply rename the legacy WAR file as a JAR file and place it in your webapp's library directory.
B. Unpack the legacy WAR file, move the TLD file to the META-INF directory, repackage the whole thing as
a JAR file, and place that JAR file in your webapp's library directory.
C. Unpack the legacy WAR file, move the TLD file to the META-INF directory, move the class files to the
top-level directory, repackage the whole thing as a JAR file, and place that JAR file in your webapp's library
directory.
D. Unpack the legacy WAR file, move the TLD file to the META-INF directory, move the class files to the
top-level directory, repackage the WAR, and place that WAR file in your webapp's WEB-INF directory.

Answer: C

Q41. Given:

```
3. public class MyTagHandler extends TagSupport {
4. public int doStartTag() {
5. // insert code here
6. // return an int
7. }
8. // more code here
...

18. }
```

There is a single attribute foo in the session scope.
Which three code fragments, inserted independently at line 5, return the value of the attribute? (Choose
three.)

A. Object o = pageContext.getAttribute("foo");
B. Object o = pageContext.findAttribute("foo");
C. Object o = pageContext.getAttribute("foo",
PageContext.SESSION_SCOPE);
D. HttpSession s = pageContext.getSession();
Object o = s.getAttribute("foo");

E. HttpServletRequest r = pageContext.getRequest();
Object o = r.getAttribute("foo");

Answer: B, C, D

Q42. You are creating a content management system (CMS) with a web application front-end. The JSP that displays a given document in the CMS has the following general structure:
1. <%-- tag declaration --%>
2. <t:document>
...
11. <t:paragraph>... <t:citation docID='xyz' /> ...</t:paragraph>
...
99. </t:document>
The citation tag must store information in the document tag for the document tag to generate a reference section at the end of the generated web page.
The document tag handler follows the Classic tag model and the citation tag handler follows the Simple
tag model. Furthermore, the citation tag could also be embedded in other custom tags that could have
either the Classic or Simple tag handler model.
Which tag handler method allows the citation tag to access the document tag?

A. public void doTag() {
JspTag docTag = findAncestorWithClass(this, DocumentTag.class);
((DocumentTag)docTag).addCitation(this.docID);
}
B. public void doStartTag() {
JspTag docTag = findAncestorWithClass(this, DocumentTag.class);
((DocumentTag)docTag).addCitation(this.docID);
}
C. public void doTag() {
Tag docTag = findAncestor(this, DocumentTag.class);
((DocumentTag)docTag).addCitation(this.docID);

}
D. public void doStartTag() {
Tag docTag = findAncestor(this, DocumentTag.class);
((DocumentTag)docTag).addCitation(this.docID);
}
Answer: A

Q43. Given:
6. <myTag:foo bar='42'>
7. <%="processing" %>
8. </myTag:foo>
and a custom tag handler for foo which extends TagSupport.

Which two are true about the tag handler referenced by foo? (Choose two.)

A. The doStartTag method is called once.
B. The doAfterBody method is NOT called.
C. The EVAL_PAGE constant is a valid return value for the doEndTag method.
D. The SKIP_PAGE constant is a valid return value for the doStartTag method.
E. The EVAL_BODY_BUFFERED constant is a valid return value for the doStartTag method.

Answer: A, C

Q44. Which two are true concerning the objects available to developers creating tag files? (Choose two.)

A. The session object must be declared explicitly.
B. The request and response objects are available implicitly.
C. The output stream is available through the implicit outStream object.
D. The servlet context is available through the implicit servletContext object.
E. The JspContext for the tag file is available through the implicit jspContext object.

Answer: B, E

Q45. You web application uses a lot of Java enumerated types in the domain model of the application. Built into each enum type is a method, getDisplay(), which returns a localized, user-oriented string. There are many uses for presenting enums within the web application, so your manager has asked you to create a custom tag that iterates over the set of enum values and processes the body of the tag once for each value; setting the value into a page-scoped attribute called, enumValue.
Here is an example of how this tag is used:

```
10. <select name='season'>
11. <t:everyEnum type='com.example.Season'>
12. <option value='${enumValue}'>${enumValue.display}</option>
13. </t:everyEnum>
14. </select>
```

You have decided to use the Simple tag model to create this tag handler.

Which tag handler method will accomplish this goal?

```
A. public void doTag() throw JspException {
try {
for ( Enum value : getEnumValues() ) {
pageContext.setAttribute("enumValue", value);
getJspBody().invoke(getOut());
}
} (Exception e) { throw new JspException(e); }
}
B. public void doTag() throw JspException {
try {
for ( Enum value : getEnumValues() ) {
getJspContext().setAttribute("enumValue", value);
getJspBody().invoke(null);
}
```

```
} (Exception e) { throw new JspException(e); }
}
C. public void doTag() throw JspException {
try {
for ( Enum value : getEnumValues() ) {
getJspContext().setAttribute("enumValue", value);
getJspBody().invoke(getJspContext().getWriter());
}
} (Exception e) { throw new JspException(e); }
}
D. public void doTag() throw JspException {
try {
for ( Enum value : getEnumValues() ) {
pageContext.setAttribute("enumValue", value);
getJspBody().invoke(getJspContext().getWriter());
}
} (Exception e) { throw new JspException(e); }
}
```

Answer: B

Q46. Given in a single JSP page:

```
<%@ taglib prefix='java' uri='myTags' %>
<%@ taglib prefix='JAVA' uri='moreTags' %>
```

Which two are true? (Choose two.)

A. The prefix 'java' is reserved.

B. The URI 'myTags' must be properly mapped to a TLD file by the web container.

C. A translation error occurs because the prefix is considered identical by the web container.

D. For the tag usage <java:tag1/>, the tag1 must be unique in the union of tag names in 'myTags' and 'moreTags'.

Answer: A, B

Q47. In a JSP-centric shopping cart application, you need to move a client's home address of the Customer object into the shipping address of the Order object. The address data is stored in a value object class called Address with properties for: street address, city, province, country, and postal code. Which two JSP code snippets can be used to accomplish this goal? (Choose two.)

A.
```
<c:set var='order' property='shipAddress'
value='${client.homeAddress}' />
```

B.
```
<c:set target='${order}' property='shipAddress'
value='${client.homeAddress}' />
```

C.
```
<jsp:setProperty name='${order}' property='shipAddress'
value='${client.homeAddress}' />
```

D.
```
<c:set var='order' property='shipAddress'>
<jsp:getProperty name='client' property='homeAddress' />
</c:store>
```

E.
```
<c:set target='${order}' property='shipAddress'>
<jsp:getProperty name='client' property='homeAddress' />
</c:set>
```

F.
```
<c:setProperty name='${order}' property='shipAddress'>
<jsp:getProperty name='client' property='homeAddress' />
```

</c:setProperty>
Answer: B, E

Q48. Given that a scoped attribute cart exists only in a user's session, which two, taken independently, ensure the scoped attribute cart no longer exists? (Choose two.)

A. ${cart = null}
B. <c:remove var="cart" />
C. <c:remove var="${cart}" />
D. <c:remove var="cart" scope="session" />
E. <c:remove scope="session">cart</c:remove>
F. <c:remove var="${cart}" scope="session" />
G. <c:remove scope="session">${cart}</c:remove>
Answer: B, D

Q49. In which three directories, relative to a web application's root, may a tag library descriptor file reside when deployed directly into a web application? (Choose three.)
A. /WEB-INF
B. /META-INF
C. /WEB-INF/tlds
D. /META-INF/tlds
E. /WEB-INF/resources
F. /META-INF/resources

Answer: A, C, E
Q50. You have been contracted to create a web site for a free dating service. One feature is the ability for one client to send a message to another client, which is displayed in the latter client's private page. Your contract explicitly states that security is a high priority. Therefore, you need to prevent cross-site hacking in which one user inserts JavaScript code that is then rendered and invoked when another user views that content. Which two JSTL code snippets will prevent cross-site hacking in the scenario above? (Choose two.)

A. <c:out>${message}</c:out>
B. <c:out value='${message}' />
C. <c:out value='${message}' escapeXml='true' />
D. <c:out eliminateXml='true'>${message}</c:out>
E. <c:out value='${message}' eliminateXml='true' />

Answer: B, C

Q51. A custom tag is defined to take three attributes. Which two correctly invoke the tag within a JSP page? (Choose two.)
A. <prefix:myTag a="foo" b="bar" c="baz" />
B. <prefix:myTag attributes={"foo","bar","baz"} />
C. <prefix:myTag jsp:attribute a="foo" b="bar" c="baz" />
D. <prefix:myTag>
<jsp:attribute a:foo b:bar c:baz />
</prefix:myTag>
E. <prefix:myTag>

<jsp:attribute ${"foo", "bar", "baz"} />
</prefix:myTag>
F. <prefix:myTag>
<jsp:attribute a="foo" b="bar" c="baz"/>
</prefix:myTag>
G. <prefix:myTag>
<jsp:attribute name="a">foo</jsp:attribute>
<jsp:attribute name="b">bar</jsp:attribute>
<jsp:attribute name="c">baz</jsp:attribute>
</prefix:myTag>
Answer: A, G

Q52. Which two are true about the JSTL core iteration custom tags? (Choose two.)
A. It may iterate over arrays, collections, maps, and strings.
B. The body of the tag may contain EL code, but not scripting code.
C. When looping over collections, a loop status object may be used in the tag body.
D. It may iterate over a map, but only the key of the mapping may be used in the tag body.
E. When looping over integers (for example begin='1' end='10'), a loop status object may not be used in the tag body.
Answer: A, C
Q53. Which two are valid and equivalent? (Choose two.)

A. <%! int i; %>
B. <%= int i; %>
C. <jsp:expr>int i;</jsp:expr>
D. <jsp:scriptlet>int i;</jsp:scriptlet>
E. <jsp:declaration>int i;</jsp:declaration>

Answer: A, E

All The Best My Dear Students

By