UNIT-7: JSP APPLICATION DEVELOPMENT

GENERATING DYNAMIC CONTENT:

Dynamic contents means the contents that get changed based on the user inputs or states of external system or on some runtime conditions. JSP helps in handling such conditions. There are various ways by which JSP handles the dynamic contents such as use of:

1. Directive elements
2. Java Beans
3. Scripting elements
4. Standard tag libraries
5. Standard and custom actions
6. Expression language.

1. Directive Elements:

Directive elements are used to specify the information about the page.

Syntax:

<%@ directivename attr1=”value1” attr2=”value2” %>

The directive names and attribute names are case sensitive.

Examples of some directives are:

- Page
- Include
- Taglib
- Attribute
- Tag
- Variable

i. Page directive:

This directive can only be used in JSP pages, not tag files. It defines page dependent attributes, such as scripting language, error page, buffer requirements.

Syntax:

<%@ Page [autoflush=”true/false”] [buffer=”skb/NNkb/None] [ContentType=”MIMEType”] [errorPage=”page or ContextRelativePath”] [extends=”classname”] [import=”packagelist”] [info=”info”][isErrorPage=”true/false”] [isThreadSafe=”true/false”] [language=”java/language”] [pageEncoding=”encoding”] [session=”true/false”] %>

Example:

<%@ page language=”java” ContentType=”text/html” %>

ii. Include directive:

It includes a static file, merging its content with the including page before the combined results is converted to JSP page implementation class.

Syntax:

<%@ include file=”page or contextRelativePath” %>

Example:

<%@ include file=”home.html” %>

iii. Taglib directive:

Declares a tag library, containing custom actions that are used in the page.

Syntax: <%@

taglib prefix=”prefix” [Uri=”tagliburi/ tagdir =”contextRelativePath”]

%>

Example:

<%@ taglib prefix=”ora” Uri=”orataglib” %>

<%@ taglib prefix=”mylib” tagdir=”/WEB-INF/tags/mylib” %>

iv. Attribute directive:

This directive can only be used in tag files. It declares the attributes that tag file supports.

Syntax:

<%@ attribute name=”attrname” [description=”desc”][required=”true/false”][fragment=”true/false”/type=”attrDataType”] %>

Example:

<%@ attribute name=”date” type=”java.util.Date” %>

v. Tag directive:

This directive can only be used in tag files.

Syntax:

<%@ tag [body-content=”empty/scriptless/tagdependent”] [description=”desc”] [display-name=”displayname”] [dynamic-attributes=”attrcolvar”] [import=”packagelist”] [language=”java/language”] [page-encoding=”encoding”] %>

Example:

<%@ tag body-content=”empty” %>

vi. Variable directive:

It is similar to variable declarations.

Syntax:

<%@ variable name-given=”attrname”/name-from-attribute=”attrname” alias=”varname” %>

Sample program for directive elements:

Scripting.jsp file:

```
<%@ page import="java.util.Date"%>

<html>

<head><title>Scripting elements example</title></head>

<body>

<%! int count=0; %>

<% count++;

out.println("You are accessing this page for "+count+ "times");

out.println(new Date().toString());

%>

<%= count %>

</body>

</html>
```

Web.xml file:

```
<web-app>

</web-app>
```

Output:

2. Java Beans:

Java beans are reusable components. We can use simple java beans in a JSP. This helps us in keeping the business logic separate from presentation logic. Beans are used in the JSP pages as instance of a class. We must specify the scope of the bean in the JSP page. Here scope of the bean means the range time span of the bean for its existence in the page.

There are various scopes using which the bean can be used in the JSP page.

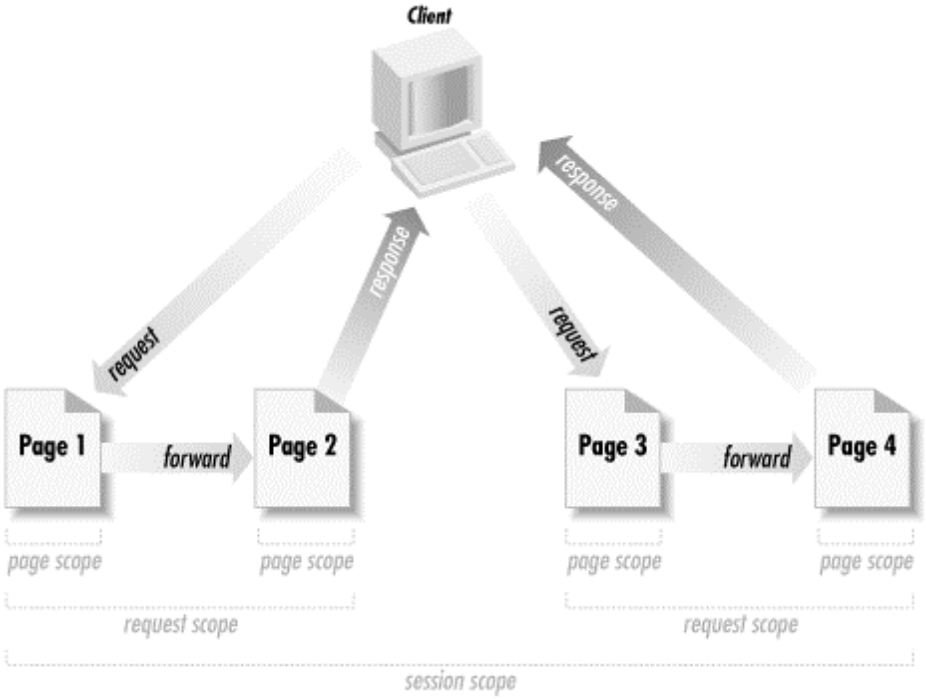
i. Page scope: The bean object gets disappeared as soon as the current page gets discarded. The default scope for a bean in the JSP page is **page** scope.

ii. Request scope: the bean object remains in the existence as long as the request object is present.

iii. Session scope: A session can be defined as the specific period of time the user spends in browsing the site. Then the time spent by the user in starting and quitting the site is one session.

iv. Application scope: during application scope the bean will gets stored to **ServletContext**. Hence particular bean is available to all the servlets in the same web application.

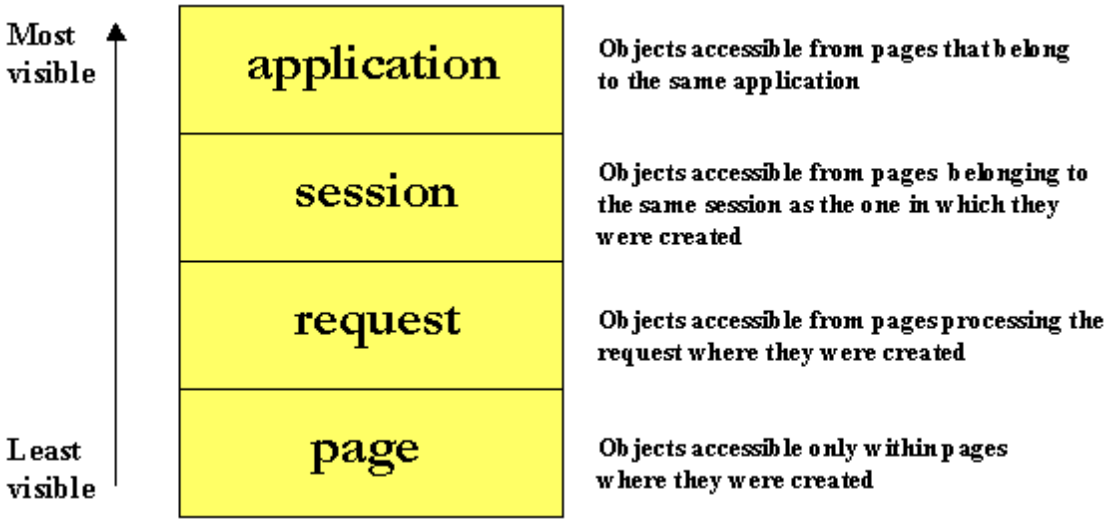
Application scope is the broadest scope provided by the JSP.



Object Scopes

Before we look at JSP syntax and semantics, it is important to understand the scope or visibility of Java objects within JSP pages that are processing a request.

Objects may be created implicitly using JSP directives, explicitly through actions,or, in rare cases, directly using scripting code. The instantiated objects can be associated with a scope attribute defining where there is a reference to the object and when that reference is removed. The following diagram indicates the various scopes that can be associated with a newly created object:



Sample program for java beans in jsp:

Getname.html file:

```
<HTML>
<BODY>
<FORM METHOD=POST ACTION="SaveName.jsp">
What's your name? <INPUT TYPE=TEXT NAME=username SIZE=20><BR>
What's your e-mail address? <INPUT TYPE=TEXT NAME=email SIZE=20><BR>
What's your age? <INPUT TYPE=TEXT NAME=age SIZE=4>
<P><INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

Nextpage.jsp file:

```
<jsp:useBean id="user" class="user.UserData" scope="session"/>
<HTML>
<BODY>
You entered<BR>
Name: <%= user.getUsername() %><BR>
Email: <%= user.getEmail() %><BR>
Age: <%= user.getAge() %><BR>
</BODY>
</HTML>
```

Savename.jsp file:

```
<jsp:useBean id="user" class="user.UserData" scope="session"/>
```

<jsp:setProperty name="user" property="*"/>

<HTML>

<BODY>

<A HREF="NextPage.jsp">Continue</A>

</BODY>

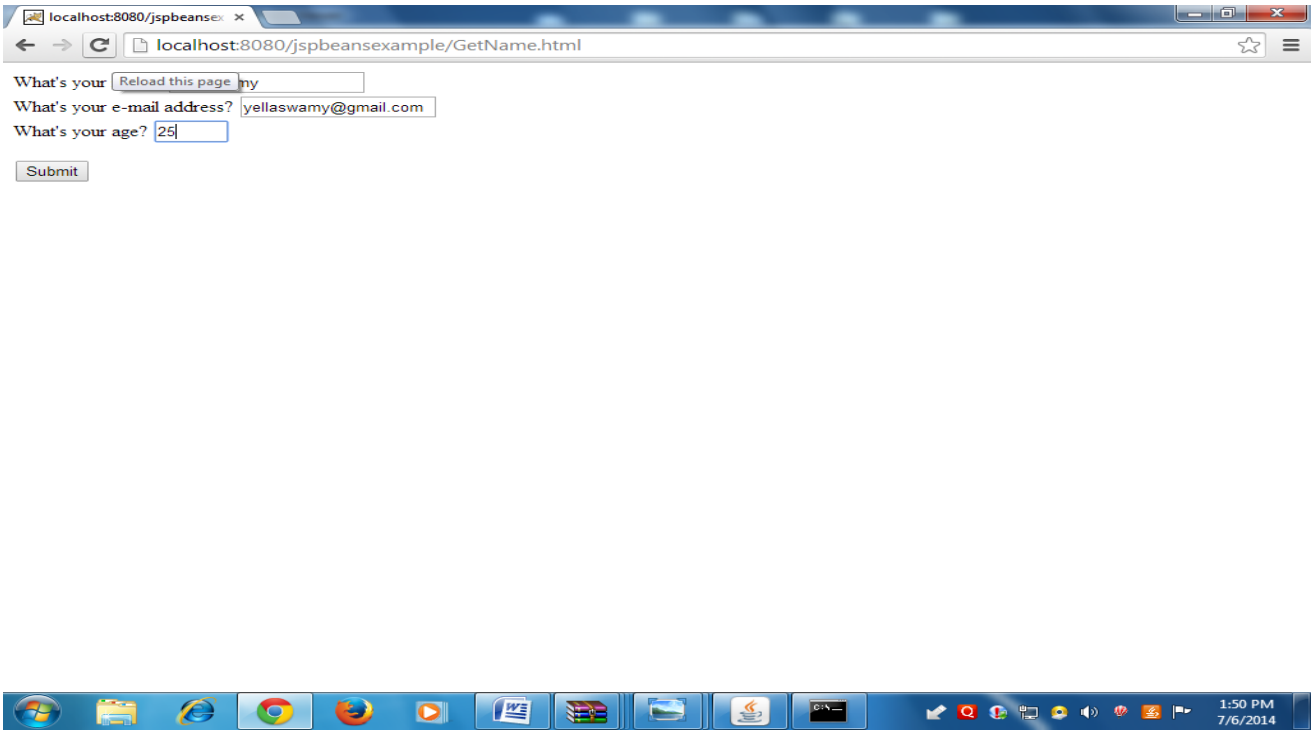
</HTML>

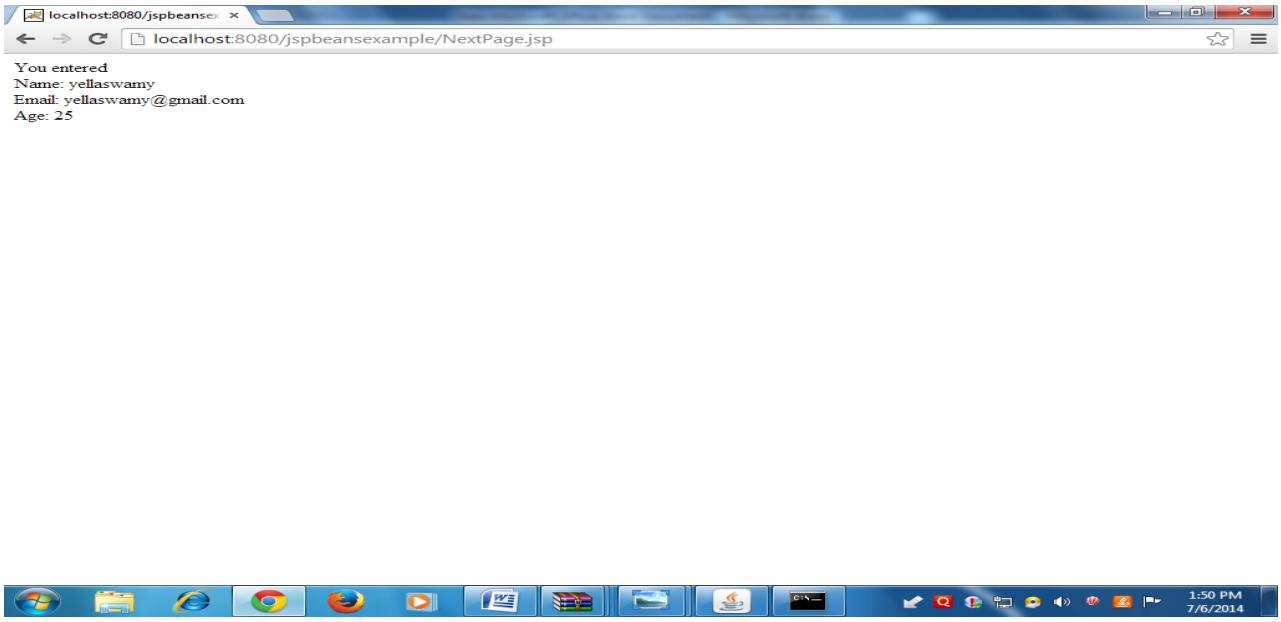
Web.xml file:

<web-app>

</web-app>

Output:





3. Scripting elements:

   Scripting allows us to embed java code in a JSP page and scripting elements (scriptlets, declaration and expressions) allows us to do scripting. Scripting elements are executed at request processing time and are used for a variety of purposes including manipulation of objects, perform calculation on runtime variables values, etc.

4. Standard tag libraries:

   JSTL stands for JSP standard tag libraries. This tag library is useful for performing some common task such as condition execution, loop execution, data processing and so on. JSTL allows the programmer to embed the logic in JSP page without using java code. The tag library makes use of some standard set of tags. To install JSTL in tomcat just copy the jstl.jar and standard.jar files in the lib folder of WEB-INF directory to your tomcat. Some tags are : <c:out>, <c:set>, <c:if>, <c:choose> etc.

5. Standard and custom actions:

   The standard actions are those actions that can be defined by the JSP specification itself. Following are some of the standard actions:

Action element	**Description**
<jsp:usebean>	This tag is used to instantiate the object of java bean
<jsp:setProperty>	This tag is used to set the property value for java bean
<jsp:getProperty>	This tag is used to get the property value from java bean
<jsp:include>	This tag works as a sub-routine. It includes the response from the servlet or JSP when the request is being processed.
<jsp:param>	For adding the specific parameter to the request this tag is used.

<jsp:forward>	This tag helps in forwarding the current request to servlet or to JSP page.
<jsp:plugin>	This tag is used to generate the HTML code and to embed the applet into it.
<jsp:attribute>	This tag sets the value of action attribute.
<jsp:element>	This tag generates the XML elements dynamically.
<jsp:text>	This tag is used to handle template text. When JSP pages are written as XML documents then this tag is used.
<jsp:body>	This tag is used to set the body element.

Custom actions allow us to create user- defined tags.

6. Expression language:

Expression language is all about generating dynamic content, without indulging in the complexity of a programming language. Hence it is used to write script-free pages. It is used by the JSP programmer in order to avoid the usage of java code for accessing data. The EL statements are always used within {…..} along with the prefix $.

IMPLICIT JSP OBJECTS:

The objects which we access in our JSP page, without any explicit declaration are called as implicit objects. Implicit objects are exposed by the JSP container and can be seen in the generated servlet of a JSP page.

Implicit objects are advantageous because, they don't require the JSP authors to explicitly declare and initialize a few of the servlet objects, which is a difficult task.

Object	**Class/Interface**	**Description**
application	javax.servlet.ServletConfig	Represents the context for the JSP page servlet, in which
session	javax.servlet.http.HttpSession	This variable is used to access current client's session.
request	javax.servlet.http.HttpServletRequest	It provides the method for accessing the information made by current request.
Response	javax.servlet.http.HttpServletResponse	It provides the methods related to adding cookies, session, setting headers.
PageContent	javax.servlet.jsp.PageContent	It provides access to several JSP attributes.
Page	Java.lang.object	This variable is assigned to instance of JSP implementation class.
Out	Javax.Servlet.jsp.JspWriter	It provides methods related to I/O.

Exception	Java.lang.Throwable	This object is used for handling error pages and contain information about runtime errors.
config	Javax.servlet.ServletConfig	It helps in passing the information to the servlet or JSP page during initialization.

Conditional processing:

Sometimes to built the complex logic we require conditional statements. We can embed JAVA conditional statements in JSP. This task can be done by scriptlets. The syntax for scriptlets is:

<% any java code %>

Sample program for conditional processing:

<%@ page language=”java” ContentType=”text/html” %>

<html>

<head><title>Introduction to scriptlet</title></head>

<body>

<h1>

<% out.println(“JSP is very interesting”); %>

</h1>

</body>

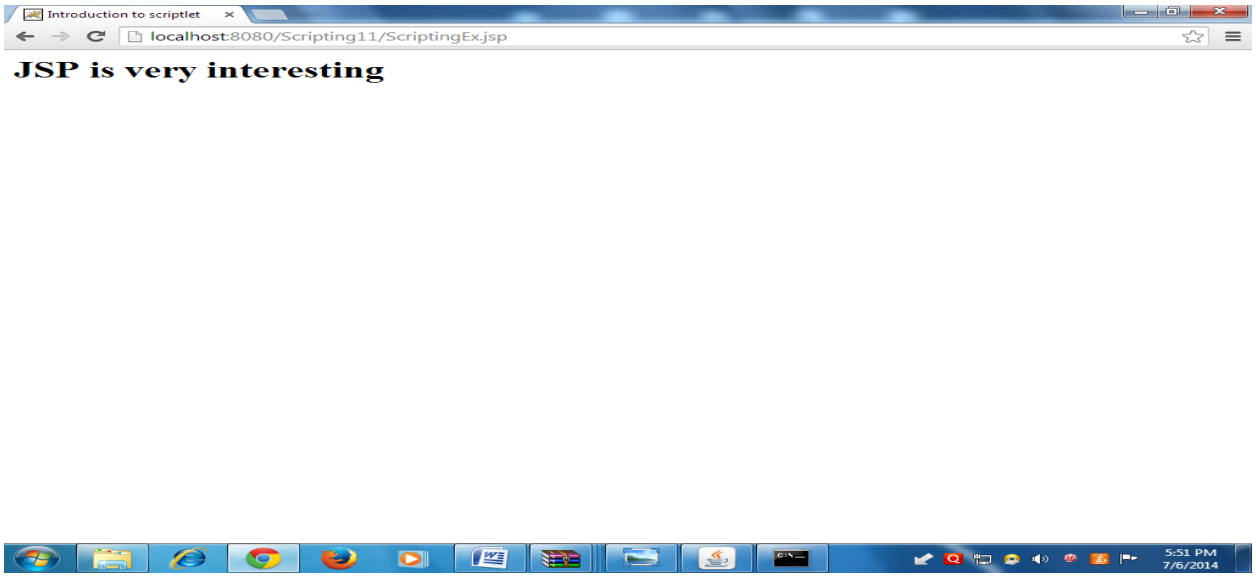
</html>

Web.xml file:

<web-app>

</web-app>

Output:



Displaying Values Using An Expression To Set An Attribute:

The JSP expressions are used to insert java values directly into the output. The syntax of using expression is as follows:

<%= Expression code %>

This expression gets evaluated at runtime and then the result will get displayed on the web browser. Thus tags <%= and %> is used.

Sample program for using an expression:

Scripting.jsp file:

```
<%@ page import="java.util.Date"%>

<html>

<head><title>Scripting elements example</title></head>

<body>

<%! int count=0; %>

<% count++;

out.println("You are accessing this page for "+count+ "times");

out.println(new Date().toString());

%>
```

<%= count %>

</body>

</html>

Web.xml file:

<web-app>

</web-app>

Directory structure:

Output:

There are some predefined variables that can be used while defining the expressions. These variables can be used along with the implicit object.

Declaring Variables And Methods:

JSP allow us to use the user defined variables and methods using declaration. The variables and methods can be used along with scriptlets and expression. The syntax for declaration is as follows:

<%! Any java code %>

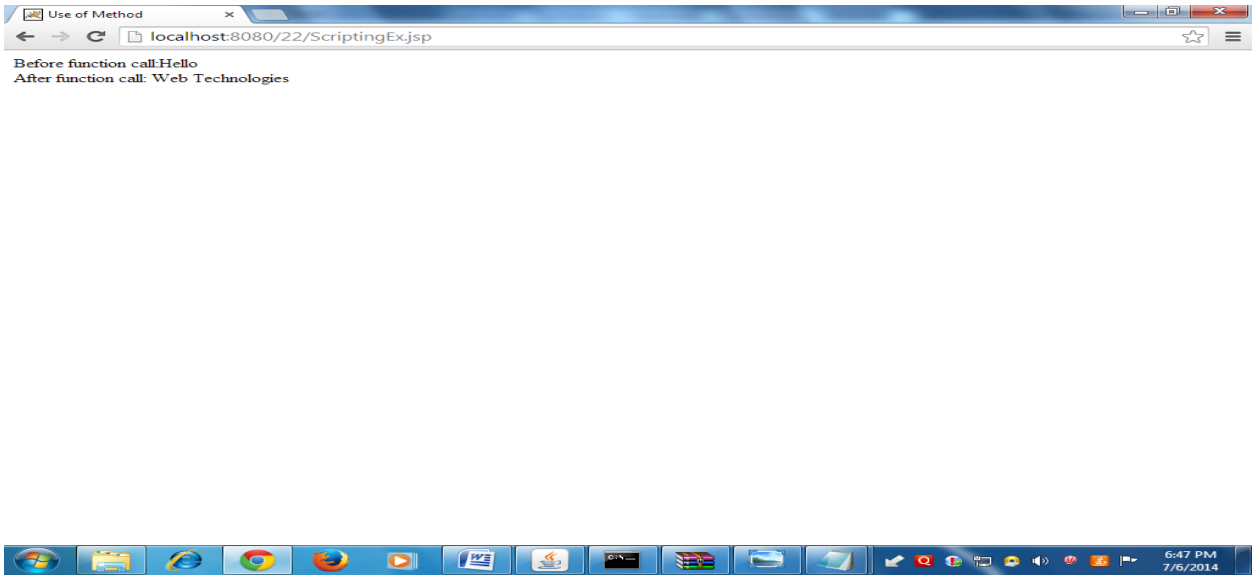
Sample program for declaring variables and methods:

```
<%@ page language="java" contentType="text/html" %>
<% String msg="Hello"; %>
<%! public String MyFunction(String msg)
{
return msg;
}
%>
<html>
<head>
<title>Use of Method</title>
</head>
<body>
<%
out.println("Before function call:"+msg); %>
<br>
After function call: <%= MyFunction("Web Technologies") %>
</body>
</html>
```

Web.xml file:

```
<web-app>
</web-app>
```

Output:



Error Handling & Debugging:

While developing any application we may come across several errors. We may get some syntactical errors during development of JSP pages.

Errors are of two types:

1. Element syntax error
2. Expression language syntax error

1. Element syntax error:

   Element syntax errors may encounter due to:

   - Improperly terminate directive.
   - Improperly terminate action
   - Mistyped attribute
   - Missing endquote in attribute values.

2. Expression language syntax error:

   Expression language syntax error may occur due to:

   - Missing both curly braces
   - Missing end curly brace
   - Misspelled property name
   - Misspelled parameter name.

Sample program for error handling:

<%@ page language=”java” contentType=”text/html”>

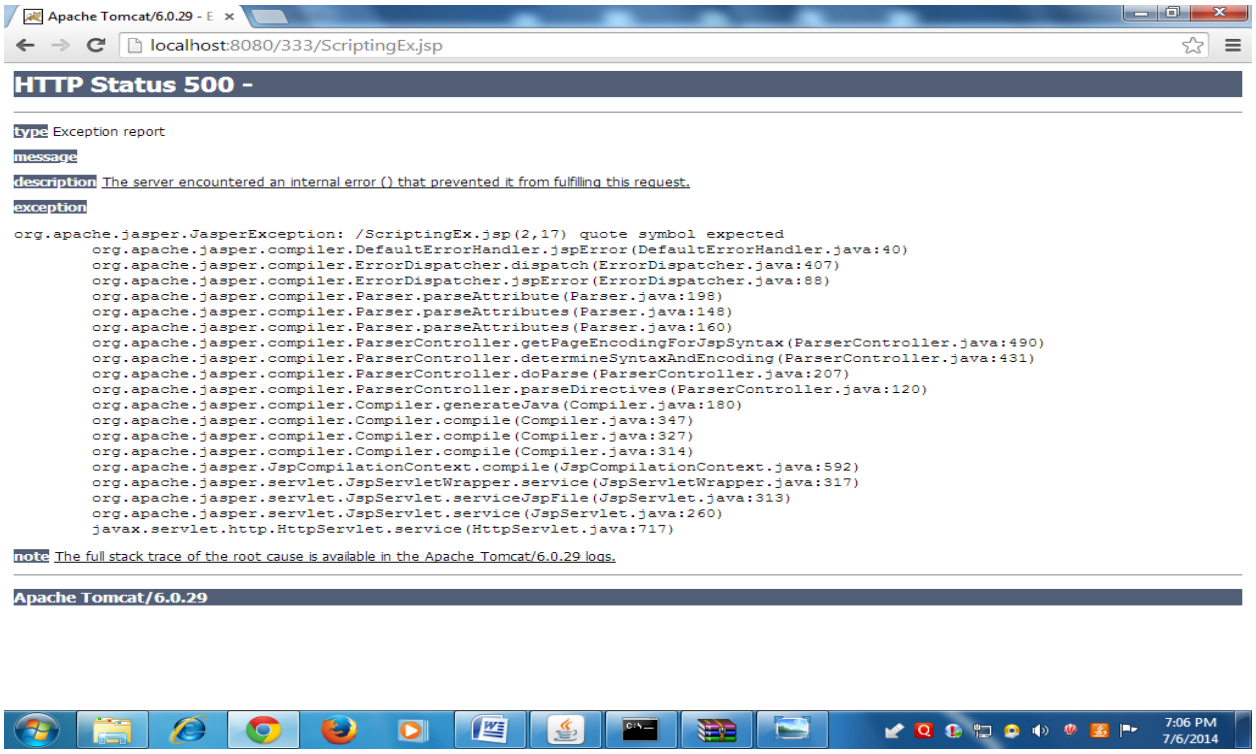
<%@ page import=”java.util.*” %>

<html>

<body>Today’s date is:

```
        <%= new Date().toString() %>
</body>
</html>
```

Error will be:



Debugging JSP actions:

Sometimes after fixing all the syntactical errors, still the application does not work properly due to logical errors. Logical errors may not be highlighted by the tomcat explicitly. The runtime errors can be handled by using exceptions and gracefully the JSP application can be terminated. Types:

- Logical error
- Dealing with runtime errors
- Catch exceptions

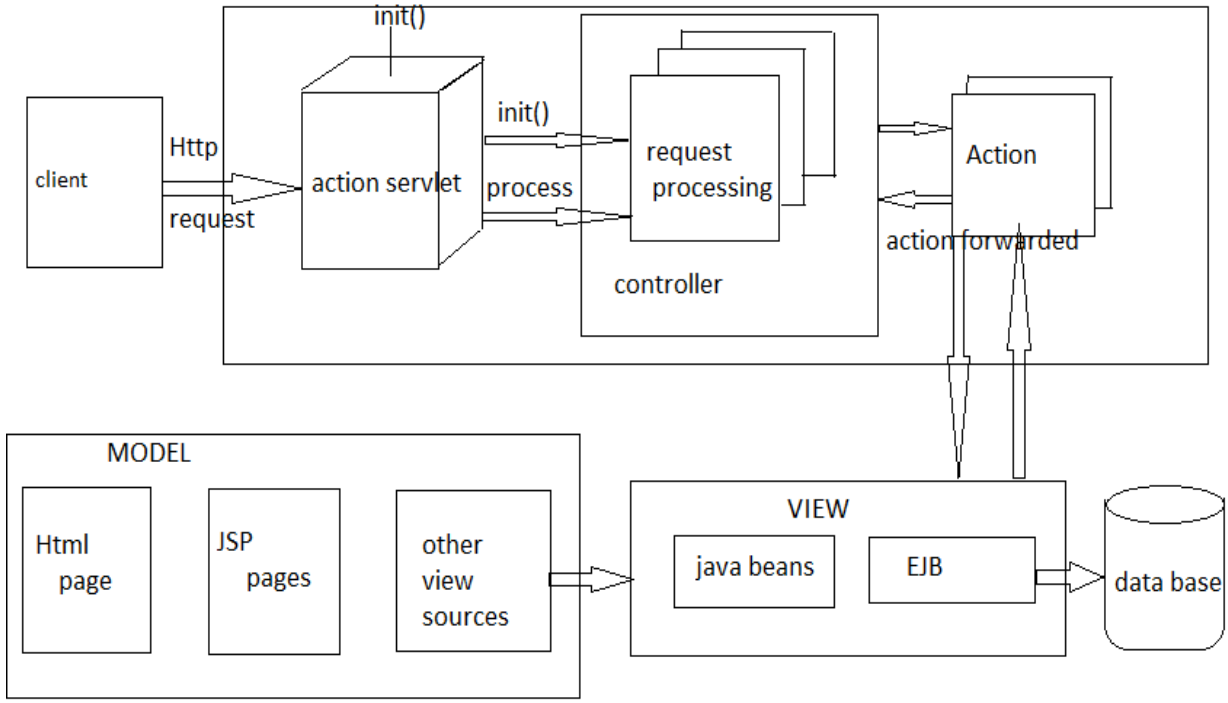
Passing Control & Data Between Pages:

Normally any web application is a client server application and it requires to handle multiple pages. These multiple pages may access same kind of information. When multiple pages access the same kind of information, the most required thing is data consistency. When particular request is being processing the data should remain same. While handling any web application JSP technologies allow some kind of partitions.

The logic is classified into three partitions:

- Request processing
- Business logic
- Presentation logic.

This three partitioned architecture is known as **"Model View Controller** (MVC) model.



We will make use of bean class in which the methods for getting and setting the counter value are written. There will be three JSP pages. On the first JSP page we will use the instance of a bean and increment the counter value, when the counter reaches the value 100 then the control will be passed to another JSP page using <jsp:forward> action. From their the control will be passed to third page by using the hyperlink used in second page.

Sharing data between JSP pages using session object:

There are some web applications in which user moves from one page to another, then it becomes necessary for data consistency. To achieve this we use an implicit object called **session.** Using session we can save the data of a particular page.

Step-1: we will create the main page on which we will accept username and password.

first_page.jsp file:

<%@ page language="java" %>

<html>

<head>

```
<title>registration form</title>
</head>
<body>
<form method=”post” action=”second_page.jsp”>
<strong>User name:</strong>
<input type=”text” name=”username”><br>
<strong>Password:</strong>
<input type=”password” name=”password”>
<input type=”submit” value=”Enter”>
</form>
</body>
</html>
```

second_page.jsp file:

```
<%@ page language=”java” %>
<%
String username=request.getParameter(“username”);
String password=request.getParameter(“password”);
session.setAttribute(“username”,username);
session.setAttribute(“password”,password);
%>
<html>
<head>
<title>session object</title>
</head>
<body>
<h3><a href=”third_page.jsp”>click here to view the other session</a>
</h3>
```

</body>

third_page.jsp file:

<%@ page language=”java” % >

<% String username=(String)session.getAttribute(“username”);

String password=(String)session.getAttribute(“password”);

%>

<html>

<head><title>end of session page</title></head>

<body>

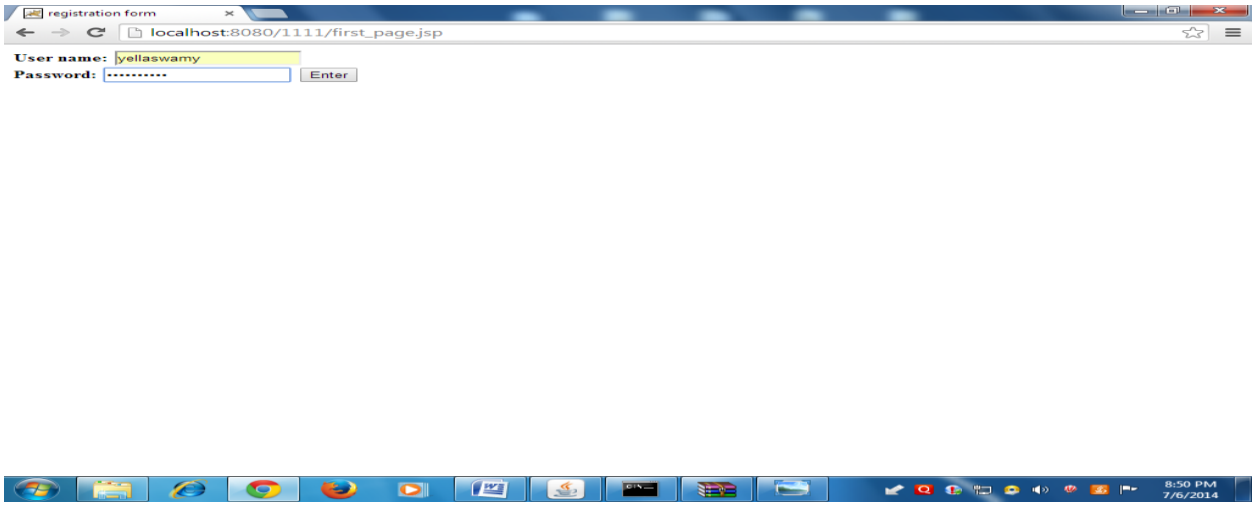
User Name is:<%= username %>

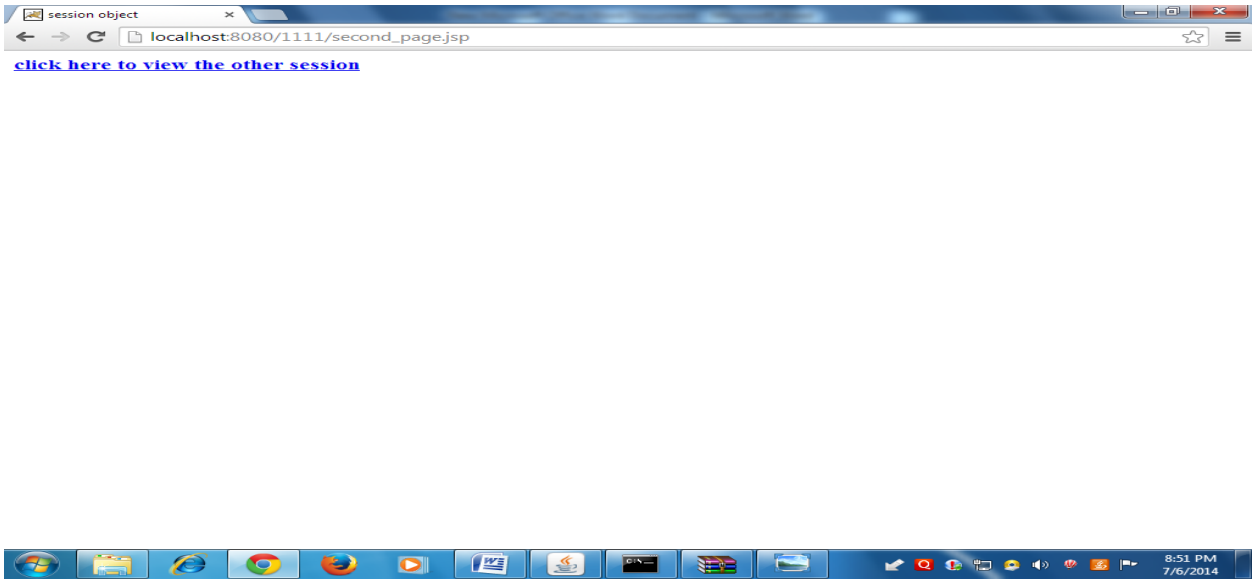
Password is:<%= password %>

</body>

</html>

Output:





Sharing data between JSP pages using application data:

Data need to be shared between different components types like servlets, filters. JSP pages when they need by any application. This is used by servlets while creating java beans and passing them to JSP pages that are responsible for displaying those beans.

Memory usage considerations:

When object gets saved in **session** scope and **application** scope they consume some memory in server process.

In application scope we can put the number of objects and by counting memory consumptions of each object. So, total memory usage can be calculated as sum of memory consumptions made by all objects. Thus we can have full control over number of objects placed in application scope.

The memory usage calculation for **session scope** is bit complex. Because total number of objects in session scope, depends upon number of concurrent sessions. Hence to compute memory usage in current session we must know certain things such as:

- Size of the object
- Number of concurrent sessions
- How long a session lasts.

Lecture

Implicit Objects

Features of Implicit Objects:

- Jsp implicit objects are used in a JSP page to make the page dynamic.
- The dynamic content can be created and accessed by using Java objects with in the scripting elements.
- JSP implicit objects are predefined objects that are accessible to all JSP Pages.
- These objects are called implicit objects because you don't need to instantiate these objects.
- The jsp container automatically instantiates these object while writing the script content in the scriplet and expression tags.

Types of Implicit objects:

During the translation of a JSP page,the JSP engine initiates Nine most commonly used implicit objects in the _jspService() method.

These JSP Implicit objects are:

S.no	Variable Name	Java Type
1	application	javax.servlet.ServletContext
2	config	javax.servlet.ServletConfig
3	exception	java.lang.Throwable
4	out	javax.servlet.jsp.JspWriter
5	page	java.lang.Object
6	PageContext	javax.servlet.jsp.PageContext
7	request	javax.servlet.http.HttpServletRequest
8	response	javax.servlet.http.HttpServletResponse
9	session	javax.servlet.http.HttpSession

Working with Implicit Objects:

This application contains the following web pages:

1. Home.html -Represent the index page of the application
2. Request.jsp -Displays the welcome message
3. PageContext.jsp -Demonstrates the use of the pageContext implicit object.
4. Other.jsp -Demonstrate the use of other implicit objects such as page,session,out,application and config.

K.Yellaswamy ,AssistantProfessor|CMR College of Engineering & Technology
E-mail:toyellaswamy@gmail.com

<!-- Home.html -->

```
<html>
 <body>
 <form action="request.jsp">
          Name : <input type="text" name="name">
          <input type="submit" value="Invoke JSP"/>
 </form>
 </body>
</html>
```

<!-- request.jsp -->

```
<html>
 <head><title>
Using Implicit Objects
 </title></head>
<body>

Hello, <b><%=request.getParameter("name")%></b><br/><br/>
Your request details are <br/><br/>
<table border="1">
<tr><th>Name</th><th>Value</th></tr>
<tr><td>request method</td>
<td><%= request.getMethod() %></td></tr>
<tr><td>request URI</td>
<td><%= request.getRequestURI() %></td></tr>
<tr><td>request protocol</td>
<td><%= request.getProtocol() %></td></tr>
<tr><td>browser</td>
<td><%= request.getHeader("user-agent") %></td></tr>
</table>
<%if (session.getAttribute("sessionVar")==null) {
      session.setAttribute("sessionVar",new Integer(0));
   }%>

<table>
<tr><th align=left>Would you like to see use of remaining imlplicit objects?</th></tr>
<tr>
<form name=form1 action="pageContext.jsp" method="post">
<td><input type="radio" name="other" value="Yes">Yes</td>
<td><input type="radio" name="other" value="No" > No</td></tr>
<tr><td><input type="submit" value="Submit"></td></tr>
```

```
</form>
</table>

</body></html>
```

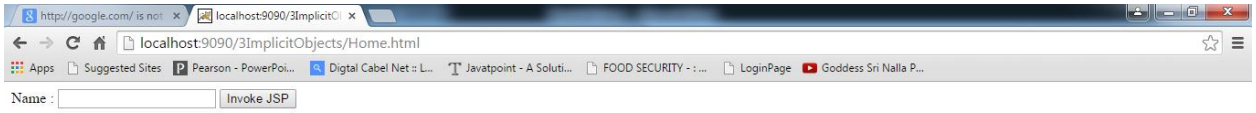
<!-- pageContext.jsp -->

```
<HTML>
 <HEAD>
  <TITLE> Intermediate </TITLE></HEAD>

 <BODY>
  <%
  if("Yes".equals(request.getParameter("other")))
 {
  pageContext.forward("other");
 }
 %>
 </BODY>
</HTML>
```

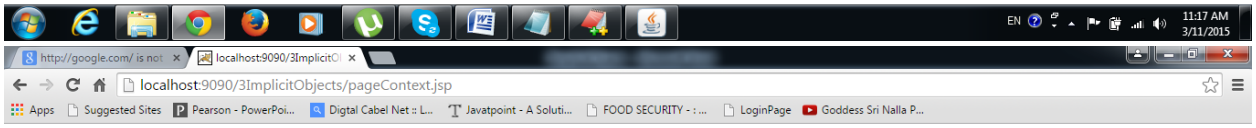
<!-- other.jsp -->

```
<html><body>
<%!int count;
        public void jspInit() {
        ServletConfig sc=getServletConfig();

        count=Integer.parseInt( sc.getInitParameter("count"));
        System.out.println("In jspInit");

        }
%>

Count value without using config implicit object: <b> <%=count%></b> <br/>
<%
this.log("log message");
((HttpServlet)page).log("anothermessage");
ServletContext ct = config.getServletContext();
out.println("Value of sessionVar
is:"+"  <b>"+session.getAttribute("sessionVar")+"</b><br/>");
```

```
out.println("Server name and version using config implicit
object:"+"  <b>"+ct.getServerInfo()+"</b><br/>");
out.println("Value of context parameter param1 get using application implicit object:"
+"  <b>"+application.getInitParameter("param1")+"</b><br/>");
out.println("Count value retrieved using config implicit object:"
+"  <b>"+config.getInitParameter("count")+"</b>");
%>

</body> </html>
```




http://google.com/ is not
Using Implicit Objects
localhost:9090/3ImplicitObjects/request.jsp?name=cmr
Apps
Suggested Sites
Pearson - PowerPoi...
Digtal Cabel Net :: L...
Javatpoint - A Soluti...
FOOD SECURITY - : ...
LoginPage
Goddess Sri Nalla P...
Hello, cmr
Your request details are
Name
Value
request method
GET
request URI
/3ImplicitObjects/request.jsp
request protocol
HTTP/1.1
browser
Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2272.3 Safari/537.36
Would you like to see use of remaining imlplicit objects?
Yes
No
Submit


EN
11:17 AM
3/11/2015
http://google.com/ is not
localhost:9090/3ImplicitO
localhost:9090/3ImplicitObjects/pageContext.jsp
Apps
Suggested Sites
Pearson - PowerPoi...
Digtal Cabel Net :: L...
Javatpoint - A Soluti...
FOOD SECURITY - : ...
LoginPage
Goddess Sri Nalla P...
Count value without using config implicit object: 10
Value of sessionVar is: 0
Server name and version using config implicit object: Apache Tomcat/6.0.29
Value of context parameter param1 get using application implicit object: param1
Count value retrieved using config implicit object: 10


EN
11:35 AM
3/11/2015

Custom Tags in JSP

Custom tags are user-defined tags. They eliminates the possibility of scriptlet tag and separates the business logic from the JSP page.

The same business logic can be used many times by the use of custom tag.

The custom tags used in the JSP Page are translated into servlet.

Advantages of Custom Tags

The key advantages of Custom tags are as follows:

1. **Eliminates the need of srciptlet tag** The custom tags eliminates the need of scriptlet tag which is considered bad programming approach in JSP.
2. **Separation of business logic from JSP** The custom tags separate the the business logic from the JSP page so that it may be easy to maintain.
3. **Reusability** The custom tags makes the possibility to reuse the same business logic again and again.
4. **Readability** The use of custom tags enhances the readability of the used codes in a JSP page.The readability increases by encapsulating the use of the Java codes in a JSP page.
5. **Maintainability** this feature enables the user to reduce the duplicity of codes present in a JSP page.

Syntax to use custom tag

There are two ways to use the custom tag. They are given below:

A)First Way
<prefix:tagname attr1=value1....attrn=valuen />

B)Second Way

<prefix:tagname attr1=value1....attrn=valuen >
body code
</prefix:tagname>

Tag Library Descriptor (TLD)

A tag library is a collection of events that encapsulate some functionality to be used from with in a JSP Page

A tag library is made available to a JSP page through a taglib directive that identifies the tag library through a URI.

Syntax:

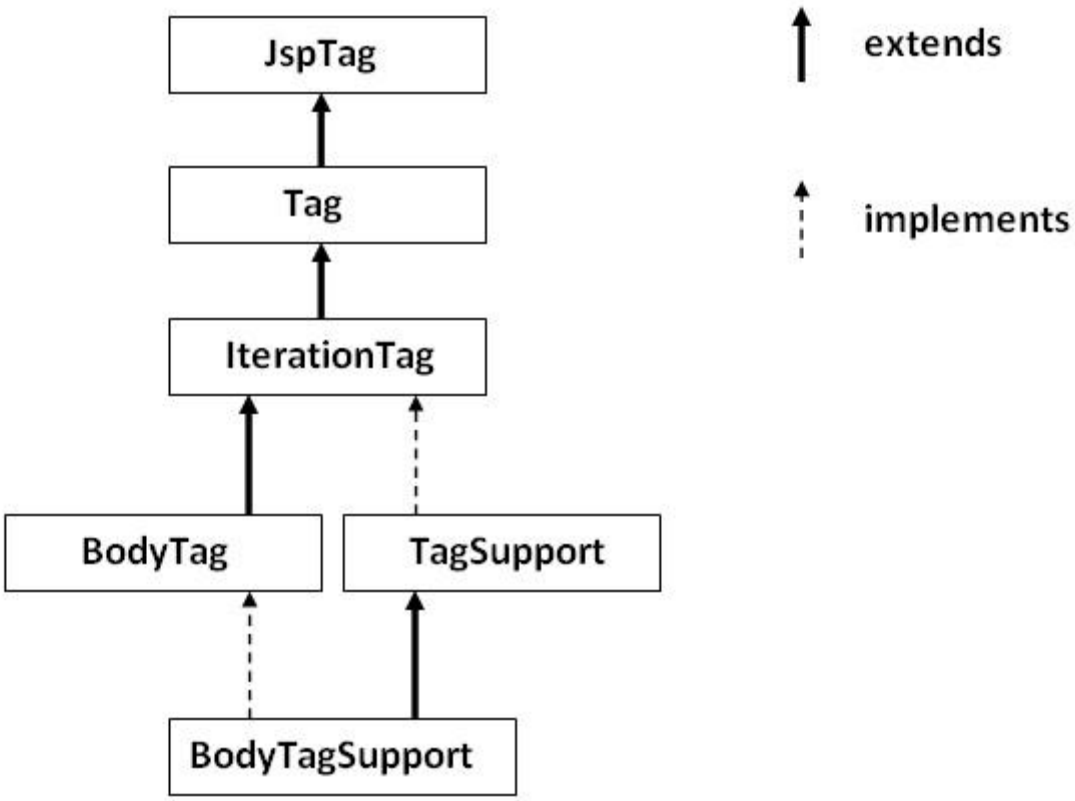
<%@taglib prefix="P" uri="example.tld"%>

Elements of TLD

description
display-name
icon
name
path
example
tag-extension
uri
tlib-version
validator
listener

JSP Custom Tag API

The javax.servlet.jsp.tagext package contains classes and interfaces for JSP custom tag API. The JspTag is the root interface in the Custom Tag hierarchy.



JspTag interface

The JspTag is the root interface for all the interfaces and classes used in custom tag. It is a marker interface.

Tag interface

The Tag interface is the sub interface of JspTag interface. It provides methods to perform action at the start and end of the tag.

Fields of Tag interface

There are four fields defined in the Tag interface. They are:

Field Name	Description
public static int EVAL_BODY_INCLUDE	it evaluates the body content.
public static int EVAL_PAGE	it evaluates the JSP page content after the custom tag.
public static int SKIP_BODY	it skips the body content of the tag.
public static int SKIP_PAGE	it skips the JSP page content after the custom tag.

Methods of Tag interface

The methods of the Tag interface are as follows:

Method Name	Description
public void setPageContext(PageContext pc)	it sets the given PageContext object.
public void setParent(Tag t)	it sets the parent of the tag handler.
public Tag getParent()	it returns the parent tag handler object.
public int doStartTag()throws JspException	it is invoked by the JSP page implementation object. The JSP programmer should override this method and define the business logic to be performed at the start of the tag.
public int doEndTag()throws JspException	it is invoked by the JSP page implementation object. The JSP programmer should override this method and define the business logic to be performed at the end of the tag.
public void release()	it is invoked by the JSP page implementation object to release the state.

IterationTag interface

The IterationTag interface is the sub interface of the Tag interface. It provides an additional method to reevaluate the body.

Field of IterationTag interface

There is only one field defined in the IterationTag interface.

- **public static int EVAL_BODY_AGAIN** it reevaluates the body content.

Method of Tag interface

There is only one method defined in the IterationTag interface.

- **public int doAfterBody()throws JspException** it is invoked by the JSP page implementation object after the evaluation of the body. If this method returns EVAL_BODY_INCLUDE, body content will be reevaluated, if it returns SKIP_BODY, no more body content will be evaluated.

TagSupport class

The TagSupport class implements the IterationTag interface. It acts as the base class for new Tag Handlers. It provides some additional methods also.

Example of JSP Custom Tag

In this example, we are going to create a **custom tag that prints the current date and time**. We are performing action at the start of tag.

For creating any custom tag, we need to follow following steps:

1. **Create the Tag handler class** and perform action at the start or at the end of the tag.
2. **Create the Tag Library Descriptor (TLD) file** and define tags
3. **Create the JSP file that uses the Custom tag defined in the TLD file**

1) Create the Tag handler class

To create the Tag Handler, we are inheriting the **TagSupport class** and overriding its method **doStartTag()**.To write data for the jsp, we need to use the **JspWriter class**.

The **PageContext** class provides **getOut()** method that returns the instance of JspWriter class. TagSupport class provides instance of pageContext bydefault.

// **MyTagHandler.java**

```
package com.cmrcet.customtags;
import java.util.Calendar;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;

public class MyTagHandler extends TagSupport
{

	public int doStartTag() throws JspException
	{
		JspWriter out=pageContext.getOut();
		try
		{
			out.print(Calendar.getInstance().getTime());
		}
		catch(Exception e)
		{
			System.out.println(e);
		}
		return SKIP_BODY;
	}

}
```

2) Create the TLD file

Tag Library Descriptor (TLD) file contains information of tag and Tag Hander classes. It must be contained inside the **WEB-INF** directory.

File: ***mytags.tld***

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE taglib
	PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
	  "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">

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

<jsp-version>1.2</jsp-version>

<short-name>simple</short-name>

<uri>http://tomcat.apache.org/example-taglib</uri>

<tag>

<name>today</name>

<tag-class>com.cmrcet.customtags.MyTagHandler</tag-class>

</tag>

</taglib>

3) Create the JSP file

Let's use the tag in our jsp file. Here, we are specifying the path of tld file directly. But it is recommended to use the uri name instead of full path of tld file.

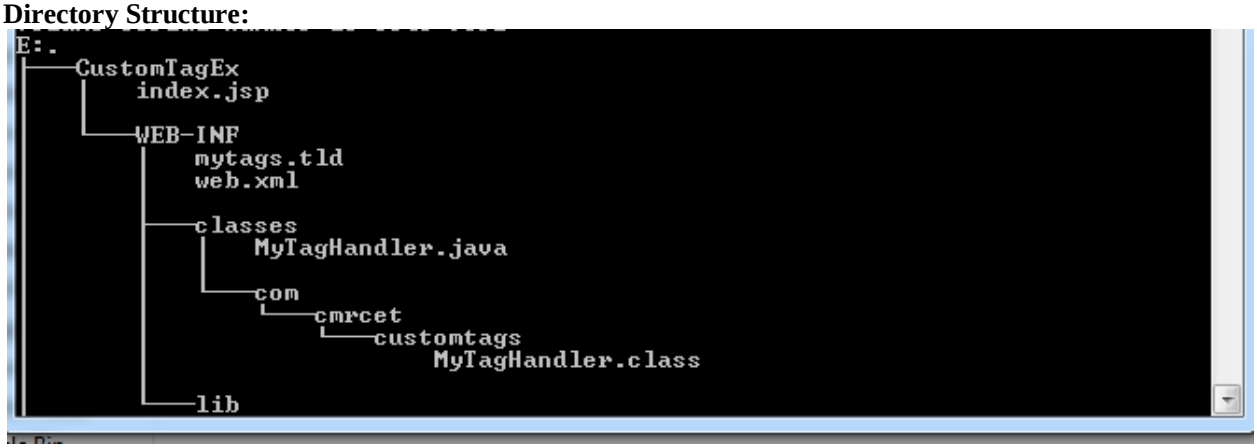
It uses **taglib** directive to use the tags defined in the tld file.

//Index.jsp

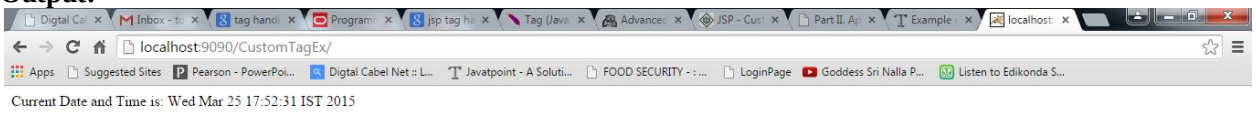
<%@ taglib uri="WEB-INF/mytags.tld" prefix="m" %>

Current Date and Time is: <m:today/>

Directory Structure:



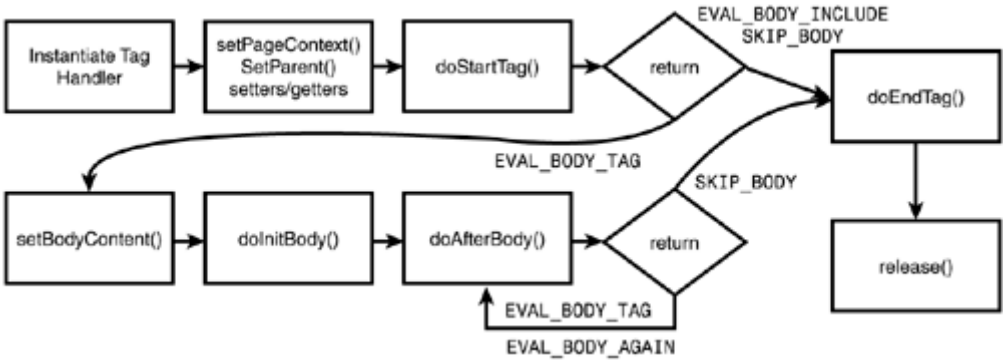
Output:



Tag Handler Life Cycle

If the container encounters a custom tags during the display of a JSP page, many events take place, beginning with the instantiation of the tag handler as shown in Figure 1

Figure1 Tag handler life cycle.



Execution of the methods implemented within tag handler classes is triggered by these events along the tag handler life cycle. The sequence of method invocation within the tag handler life cycle is as follows:

- Once a JSP tag is encountered, the methods `setPageContext()` and `setParent()` are invoked to set up an environment context for the tag handler. These methods don't have to be implemented if you're using the tag support abstract classes (`TagSupport` and `BodyTagSupport`) for development (as opposed to the tag interfaces).
- Next, the setter methods for tag attributes are invoked. Tag handlers must define setters and getters for all attributes.
- Next, the `doStartTag()` method is invoked. This method is commonly used to initialize any resources for execution of your tag logic. This method should return one of three different values depending on the nature of your tag body, assuming that a tag body is implemented:
    - `Tag.SKIP_BODY` should be returned if you're implementing a simple (empty body) tag. On return of this value, the `doEndTag()` method is invoked.
    - Returning `Tag.EVAL_BODY_INCLUDE` causes evaluation and inclusion of tag body contents. Only classes implementing the `Tag` interface or extending the `TagSupport` class may return this value. On return of this value, the `doEndTag()` method is invoked.
    - `Tag.EVAL_BODY_TAG` causes evaluation of the tag body, followed by invocation of the `doInitBody()` method. Only classes implementing the `BodyTag` Interface or extending the `BodyTagSupport` class may return this value.
- If `EVAL_BODY_TAG` is returned, the `setBodyContent()` method is invoked. This method creates a reference to the `BodyContent` (a JspWriter) buffer, which may be used by the `doAfterBody()` method for processing later in the life cycle. The `BodyContent` buffer contains all output from the tag, including any body content, if any. (At this point, the client does not have access to any output derived from the tag). If the tag is writing output to the JSP page, that output must be written to the parent-scoped JspWriter (via the `getEnclosingWriter()` method) before the end of the `doEndTag()` method. You don't have to implement the `setBodyContent()` method if you're implementing the `BodyTagSupport` class. This class has the convenience of the `getBodyContent()` method, which provides a reference to `BodyContent`.
- If `EVAL_BODY_TAG` is returned, the `doInitBody()` method is invoked. This method is invoked immediately prior to the body tag being evaluated for the first time. This method is commonly used to prepare scripting variables or place content into the `BodyContent` JspWriter. As discussed earlier, content placed into the `BodyContent` is buffered and isn't available to the JSP page at this point.
- If `EVAL_BODY_TAG` is returned, the `doAfterBody` method is invoked. This method is invoked immediately after the tag body is evaluated and added to the `BodyContent`. This method is commonly used to process work based on the results of the body tag evaluation. To access the evaluated body, use the `getBodyContent()` method if you extended the `BodyTagSupport` class, or use the `setBodyContent()` method if you implemented the `BodyTag` interface (you should have stored a `BodyContent` instance).

NOTE

The JSP 1.2 spec provides a `javax.servlet.jsp.tagext.IterationTag` interface, which enables you to reprocess the tag body if desired. This reprocessing would most likely be driven by a preset condition, much like iteration or loop logic.

The `doAfterBody()` method returns one of two different values depending on the nature of your tag body:

- `Tag.SKIP_BODY` should be returned if no further processing of the tag body is required. Upon return of this value, the `doEndTag()` method is invoked.
- `Tag.EVAL_BODY_TAG (IterationTag.EVAL_BODY_AGAIN)` causes evaluation of the tag body again. The result of the tag body evaluation is appended to the `BodyContent`. The `doAfterBody()` method is invoked again.

To write to the surrounding scope, you can obtain a writer using method `BodyTagSupport.getPreviousOut()` ormethod `BodyContent.getEnclosingWriter()` as discussed earlier. Both methods return the same JspWriter. Because the `BodyContent` is appended on each iteration through the tag body, you should write to this surrounding scope only when you've decided to return `Tag.SKIP_BODY`.

At this point in the life cycle, the writer in the `pageContext()` is returned to the parent `JspWriter`.

Next, the `doEndTag()` method is invoked. This method is commonly used to perform post–body tag processing, close server-side resources, or to write output to the surrounding scope using the `pageContext.getOut()` method.

The `doEndTag()` method returns one of two different values depending on the nature of your page:

- `Tag.EVAL_PAGE` causes evaluation of the remaining JSP page.
- `Tag.SKIP_PAGE` terminates evaluation of the remaining JSP page.

To conclude the tag handler life cycle, the `release()` method is invoked just before the tag handler instance is made available for garbage collection.