Jatin Ambasana

SUBJECT:
Advanced Java Programming

TOPIC:
Java Server Pages

Outline

1. Introduction
2. Comparison with Servlets
   - JSP Vs Servlets
3. First JSP Program
4. JSP Architecture
   - Life Cycle of JSP
5. JSP Scripting Elements
6. JSP Directives
7. JSP Action tags
8. JSP Implicit Objects
9. Expression Language
10. JSP Standard Tag Libraries
    1. JSTL Tags
       1. Core
       2. Function
       3. SQL
    2. JSP XML Tag
11. Custom Tag
12. Session Management
    - Cookies handling
13. Exception Handling
14. JSP application design with MVC
15. Advantages of MVC Design
16. Form processing
17. Database access

INTRODUCTION

Introduction

1. JSP Stands for **Java Server Pages**.
2. A JSP page consists of **HTML tags and JSP tags.**
3. The JSP pages are easier to maintain than servlet because we can **separate designing and development logic.**
4. JSP is a **server side scripting language**.

COMPARISON WITH SERVLETS

Problems with Servlet

- Inside the one and only one class in servlet alone, we will perform various task:
  - Acceptance of request
  - Process the request
  - Handling the business logic
  - Generation of response
- For creating a servlet **knowledge of java and html both is needed.**
- While developing an app if the look and feel of prog. changes we have to **change the entire servlet class.**

JSP Vs Servlets

Sr. No.	JSP	SERVLET
1	**JSP is a scripting lang** which generates dynamic content	**Servlet are java prog.** that can be compiled to generate dynamic content
2	**JSP runs slower** than servlet	**Servlet runs faster** than JSP
3	In MVC **JSP acts as view.**	In MVC **Servlet acts as controller.**
4	JSP **can build custom tags**	**No facility** for creating custom tags
5	It is easier to code a JSP	The servlet are basically complex java prog

Prog: First JSP Program

- Any text, HTML tags, or JSP elements you write must be outside the scriptlet. Following is the simple and first example for JSP:

```
<html>
<head><title>Hello World</title></head>
<body>
    Hello World! <br/>
<%
    out.println("Hello World!");
%>
</body>
</html>
```

JSP SCRIPTING ELEMENTS

JSP SCRIPTING ELEMENTS

Scriptlets

- A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language.
- Following is the syntax of Scriptlet:

```
<% code fragment %>
```

- There are three types of scripting elements:
- scriptlet tag
- expression tag
- declaration tag

Scriptlet Tag

- In JSP, **java code can be written** inside the jsp page using the scriptlet tag.
- Syntax:

```
<%

      ……..

                  …….. .     …….. .

%>
```

Scriptlet Tag Example

- Example:

```
<html>
<head><title>Hello World</title></head>
<body>
Hello World!<br/>
<%
    out.println("Your IP address is " +
                request.getRemoteAddr());
%>
</body>
</html>
```

Prog: JSP scriptlet tag that prints the user name

- File1: index.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
</body>
</html>
```

Prog: JSP scriptlet tag that prints the user name

- File2. welcome.jsp

```
<html>
<body>
<%
 String name=request.getParameter("uname");
 out.print("welcome "+name);
%>
</body>
</html>
```

JSP expression tag

- The code placed within JSP expression tag is written to the output stream of the response.
- Syntax of JSP expression tag:

```
<%=  statement %>
```

JSP expression tag (Cont.)

- A JSP expression element contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file.

- Because the value of an expression is converted to a String, you can use an expression within a line of text, whether or not it is tagged with HTML, in a JSP file.

Example 1 of JSP expression tag

- In this example of jsp expression tag, we are simply displaying a welcome message.

```
<html>
<body>
  <%= "welcome to jsp" %>
  Today's date:
  <%= (new java.util.Date() %>
</body>
</html>
```

Example 2 of JSP expression tag that prints current time

```
<html>
<body>
Current Time:
<%=
   java.util.Calendar.getInstance().getTime()
%>
</body>
</html>
```

Example 3 of JSP expression tag that prints the user name

- File: index.jsp

```
<html>
<body>
   <form action="welcome.jsp">
   <input type="text" name="uname"><br/>
   <input type="submit" value="go">
</form>
 </body>
</html>
```

Example 3 of JSP expression tag that prints the user name

- File: welcome.jsp

```
<html>
<body>
<%=
   "Welcome "+request.getParameter("uname")
%>
</body>
</html>
```

JSP Declaration Tag

- The JSP declaration tag is used to declare fields and methods.
- Syntax of JSP declaration tag
- The syntax of the declaration tag is as follows:

```
<%!  field or method declaration %>
```

- Following is the simple example for JSP Declarations:

```
<%! int i = 0; %>
<%! int a, b, c; %>
<%! Circle a = new Circle(2.0); %>
```

Example 1 of JSP declaration tag that declares field

- File: index.jsp

```
<html>
<body>
    <%! int data=50; %>
    <%= "Value of the variable is:"+data %>
</body>
</html>
```

Example 2 of JSP declaration tag that declares method

- File: index.jsp

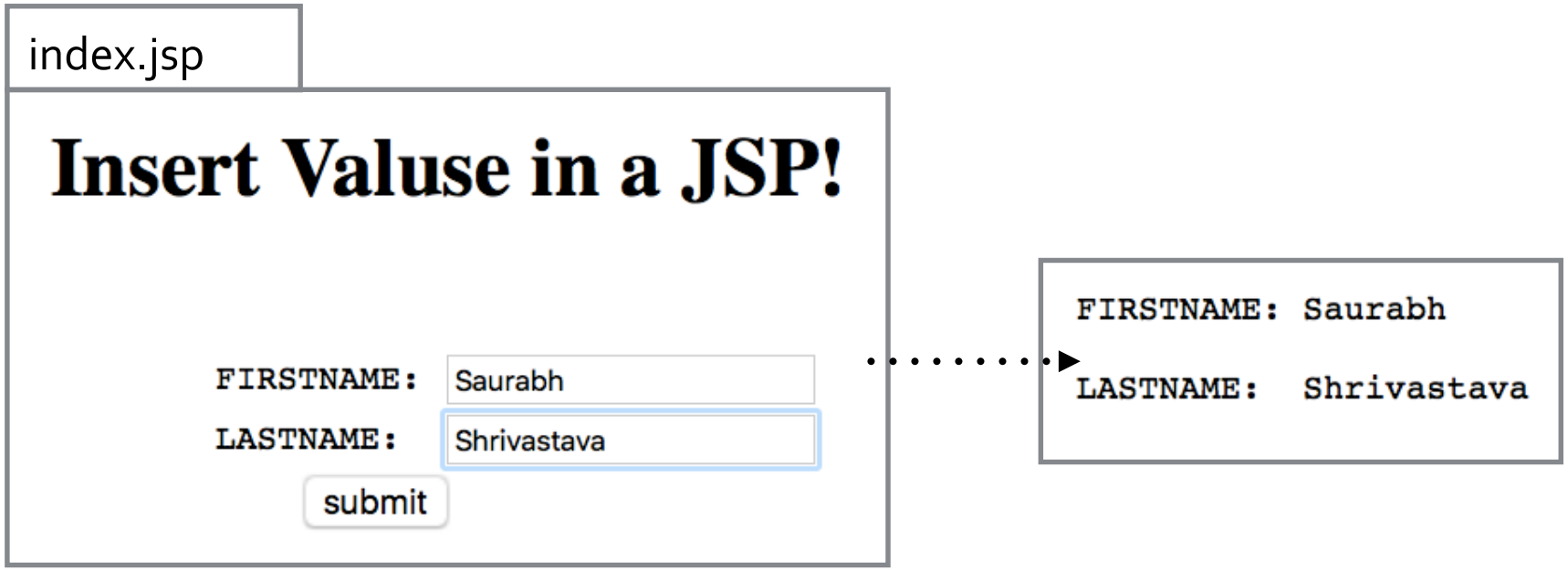
```
<html>
<body>
    <%!
        int cube(int n){
        return n*n*n*;
    }
    %>
    <%= "Cube of 3 is:"+cube(3) %>
</body>
</html>
```

SIMPLE FORM PROCESSING EXAMPLE

JSP FORM PROCESSING

1. Form is a **very common way of interaction** in web sites.
2. The **<form>** tag is used so that user can enter the form details.
3. The web-browser used two method to pass this data to business logic
4. The two methods are **GET and POST.**

Prog: JSP Form Processing

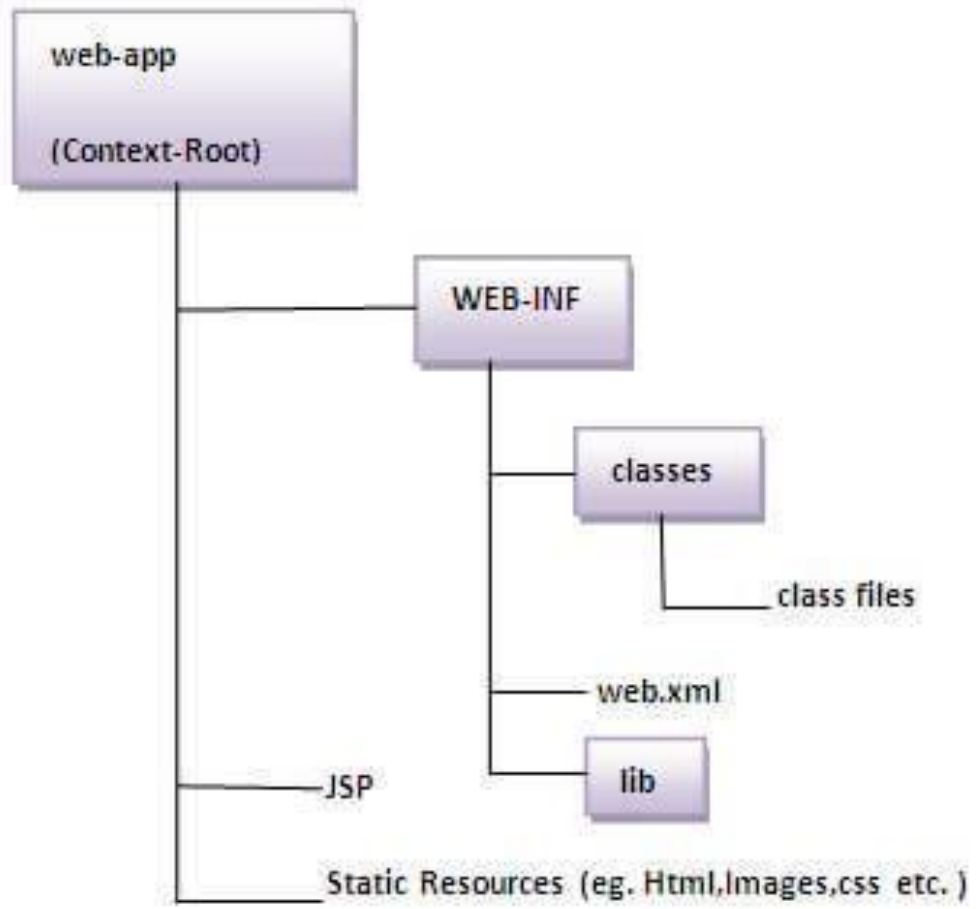


Files needed to be created:

- index.jsp
- demojsp.jsp
- web.xml

The Directory structure of JSP

- The directory structure of JSP page is same as Servlet. We contain the JSP page outside the WEB-INF folder or in any directory.



Files needed to be created:

- File: index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<body>
      <form action= "demojsp.jsp" method="GET">
            FIRSTNAME:  <input type="text" name="fname"/>
            LASTNAME:   <input type="text" name="lname"/>
                        <input type="submit" value="submit"/>
      </form>
</body>
</html>
```

Files needed to be created:

- File: demojsp.jsp

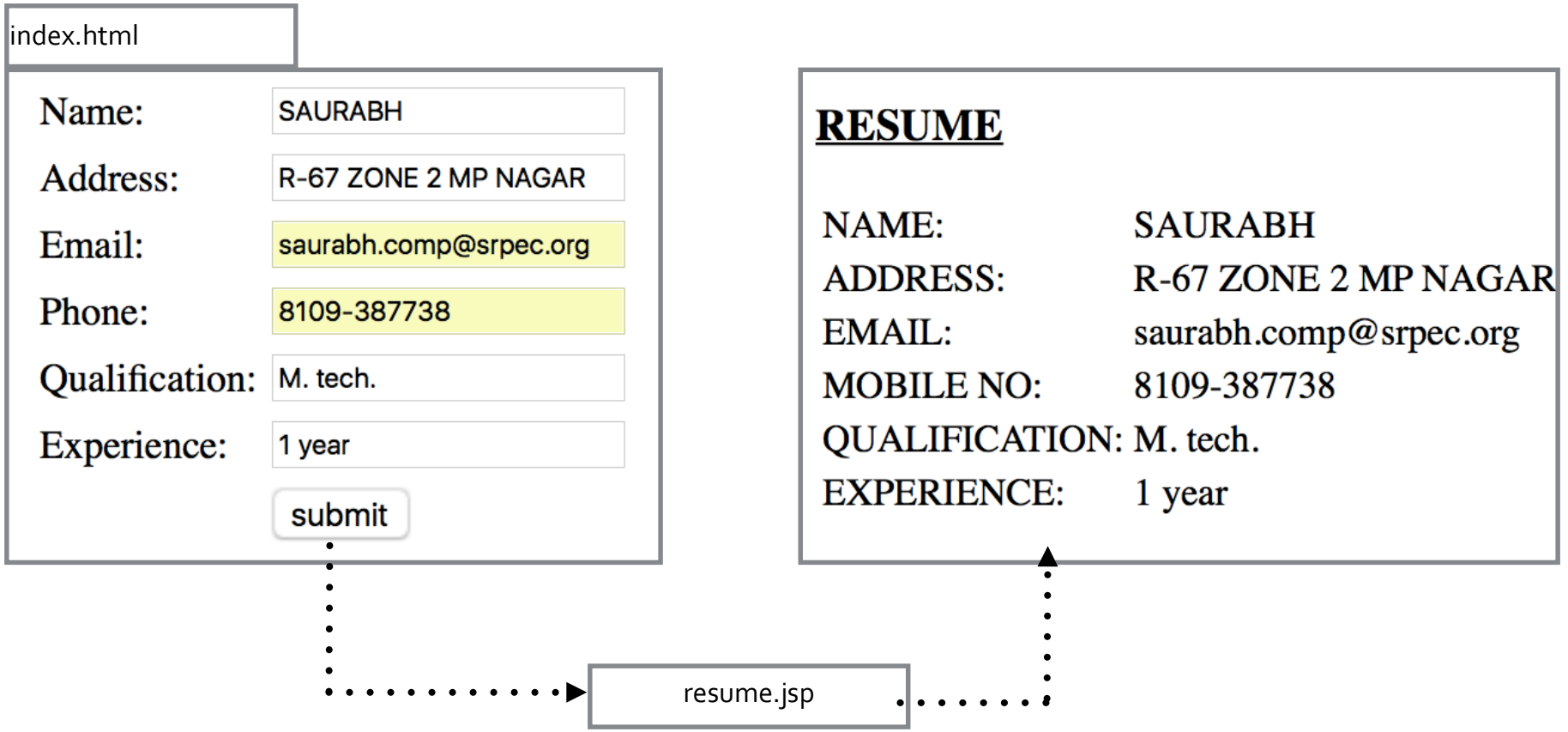
```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <body>
     FIRSTNAME: <%= request.getParameter("fname") %> <br>
     LASTNAME:  <%= request.getParameter("lname") %>
    </body>
</html>
```

Files needed to be created:

- File: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app >
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

TASK: Write a prog that prints your details through a Form

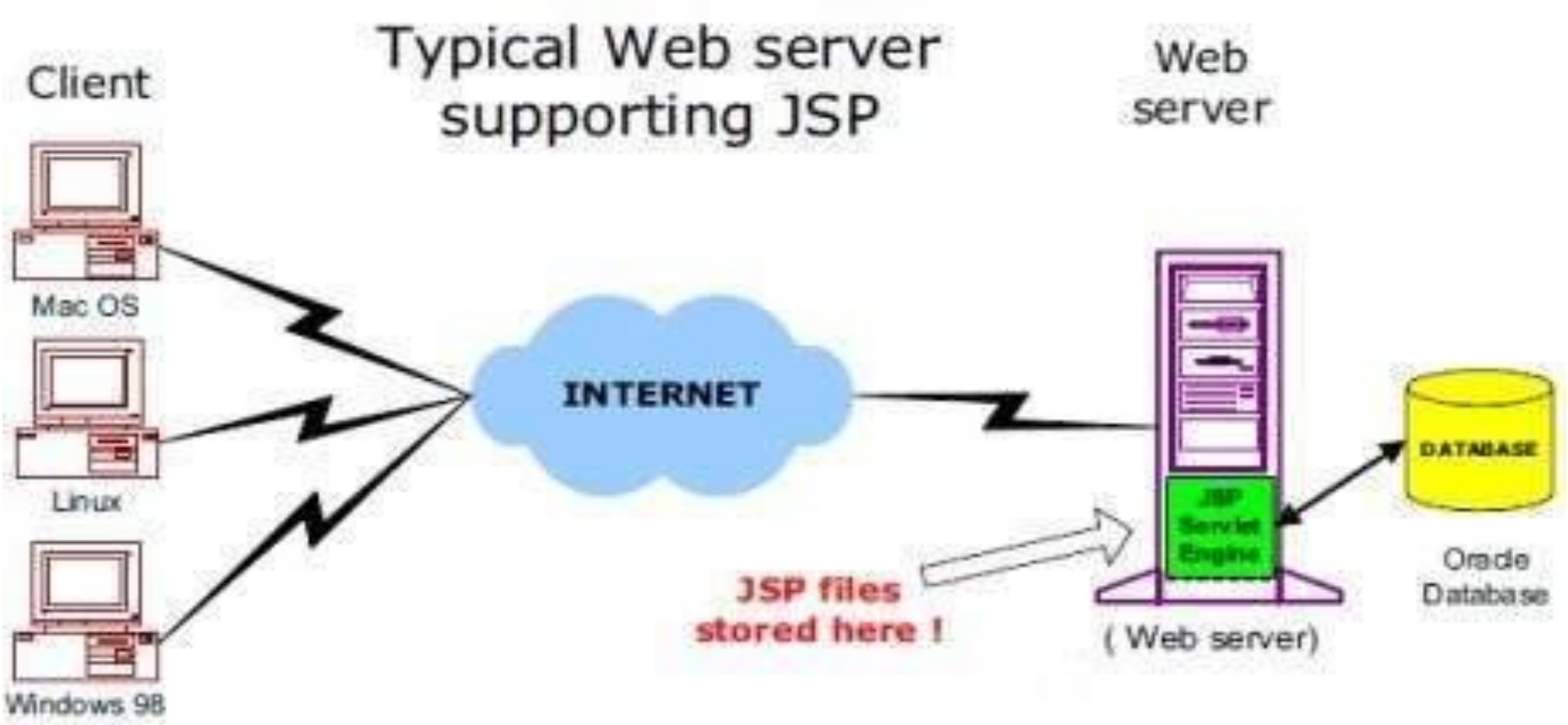


JSP ARCHITECTURE

JSP - Architecture

- The web server needs a JSP engine ie. container to process JSP pages.
- The JSP container is responsible for intercepting requests for JSP pages.
- A JSP container works with the Web server to provide the runtime environment and other services a JSP needs. It knows how to understand the special elements that are part of JSPs.

JSP – Architecture (Cont.)



JSP Processing

- As with a normal page, your browser sends an HTTP request to the web server.

- The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with **.jsp** instead of .html.

- The JSP engine loads the JSP page from disk and converts it into a servlet content.

JSP Processing (Cont.)

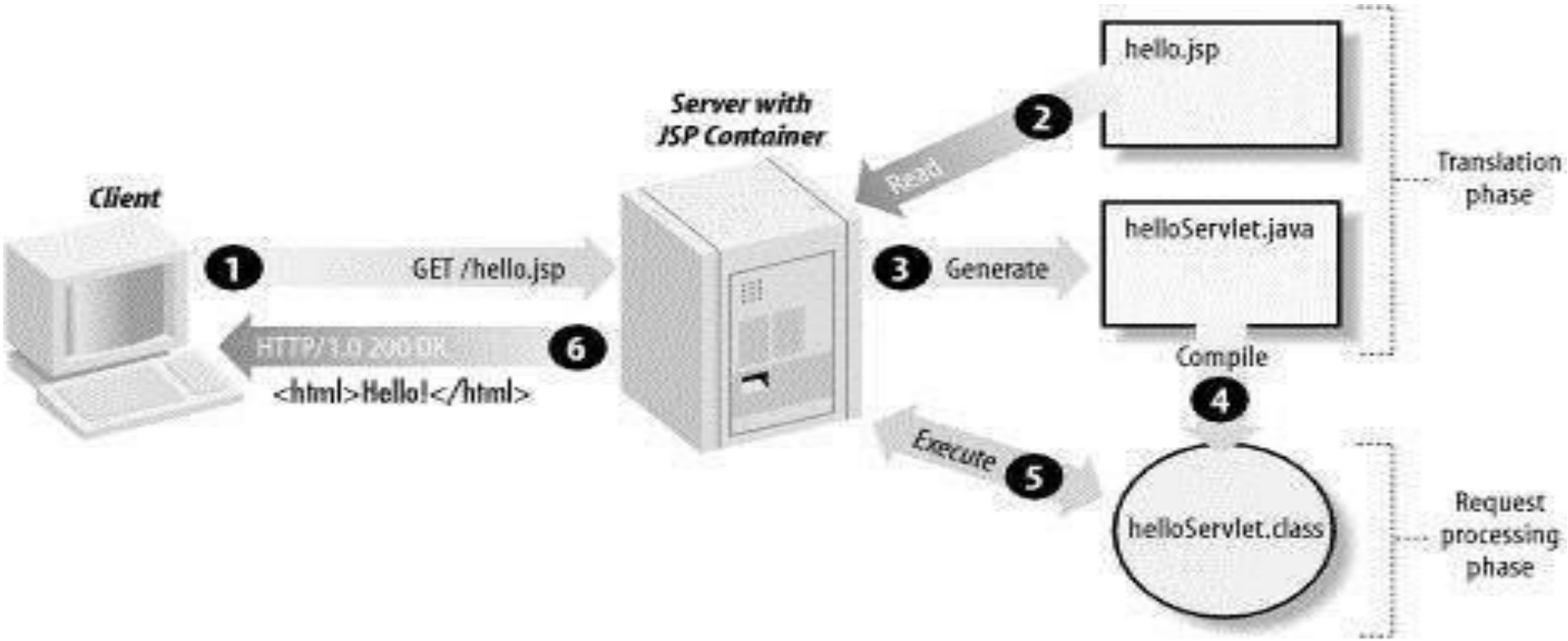
- This conversion is very simple in which all template text is converted to println( ) statements and all JSP elements are converted to Java code that implements the corresponding dynamic behavior of the page.

- The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.

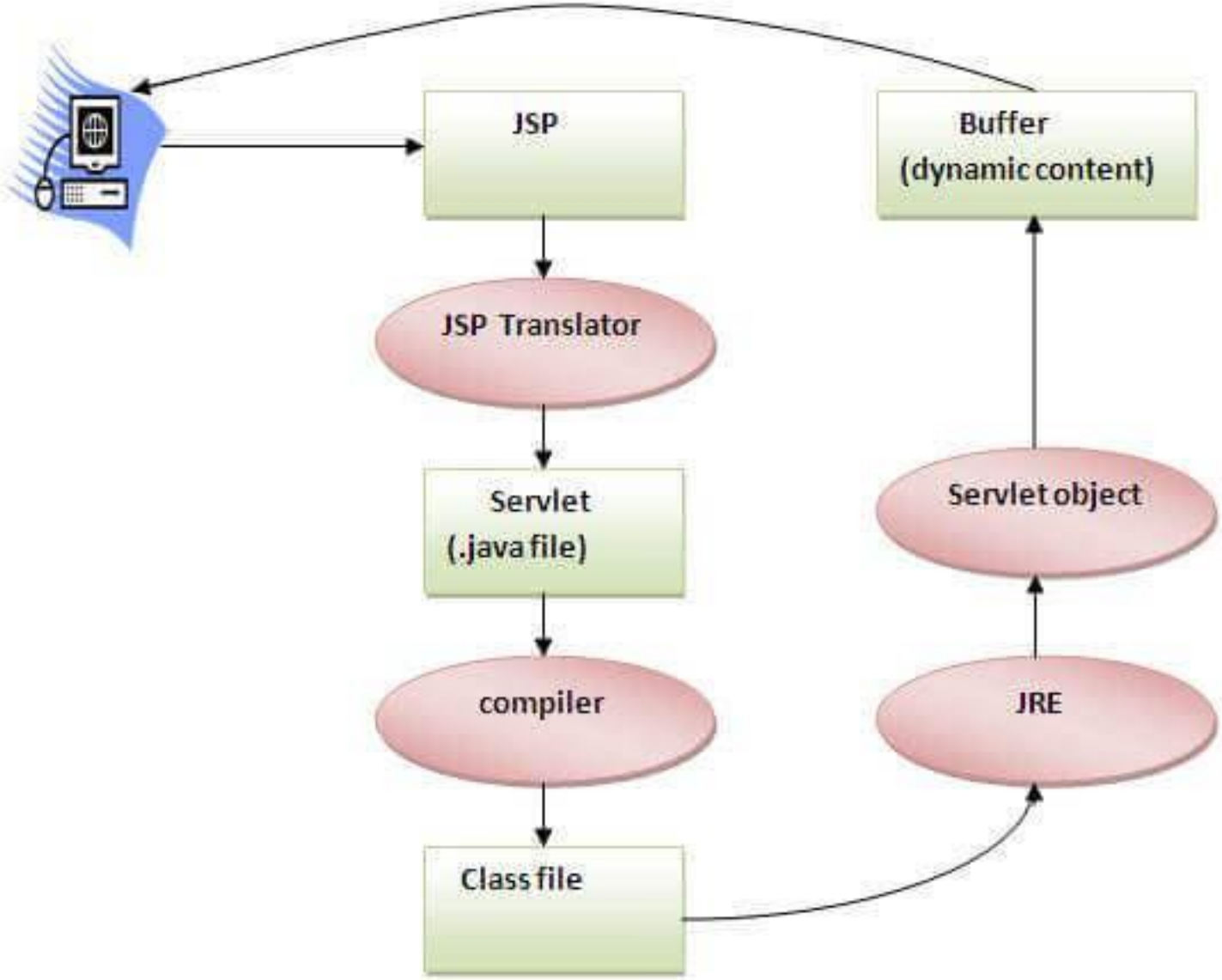
- A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format, which the servlet engine passes to the web server inside an HTTP response.

JSP Processing (Cont.)

- The web server forwards the HTTP response to your browser in terms of static HTML content.
- Finally web browser handles the dynamically generated HTML page inside the HTTP response exactly as if it were a static page.

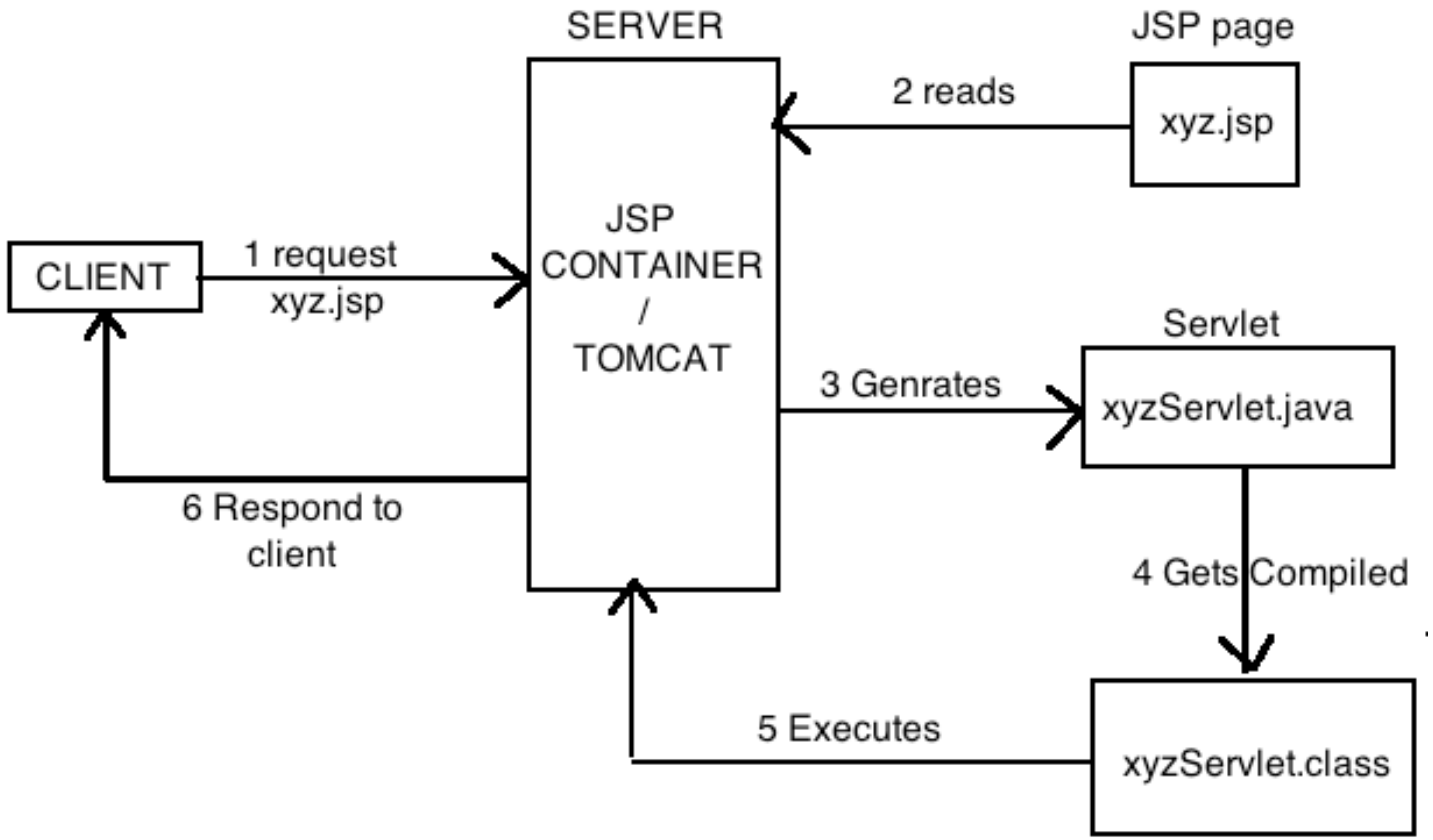


LIFE CYCLE OF JSP (cont.)



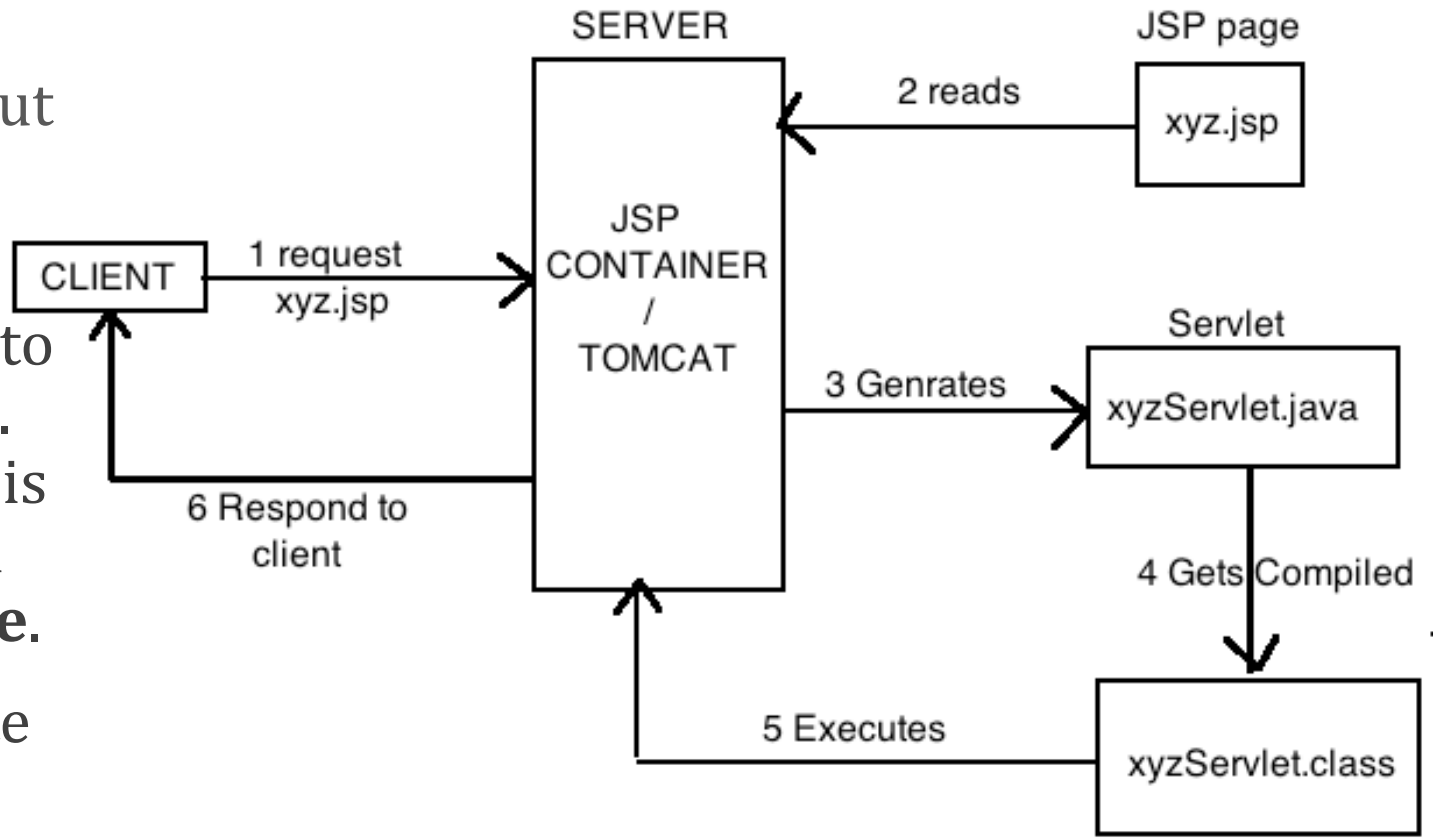
LIFE CYCLE OF JSP

1. Client makes a request for required JSP page to the server.
2. The server must have a JPS container to process the request.
3. For the request the JSP container searches the request and reads the desired JSP page.



LIFE CYCLE OF JSP (cont.)

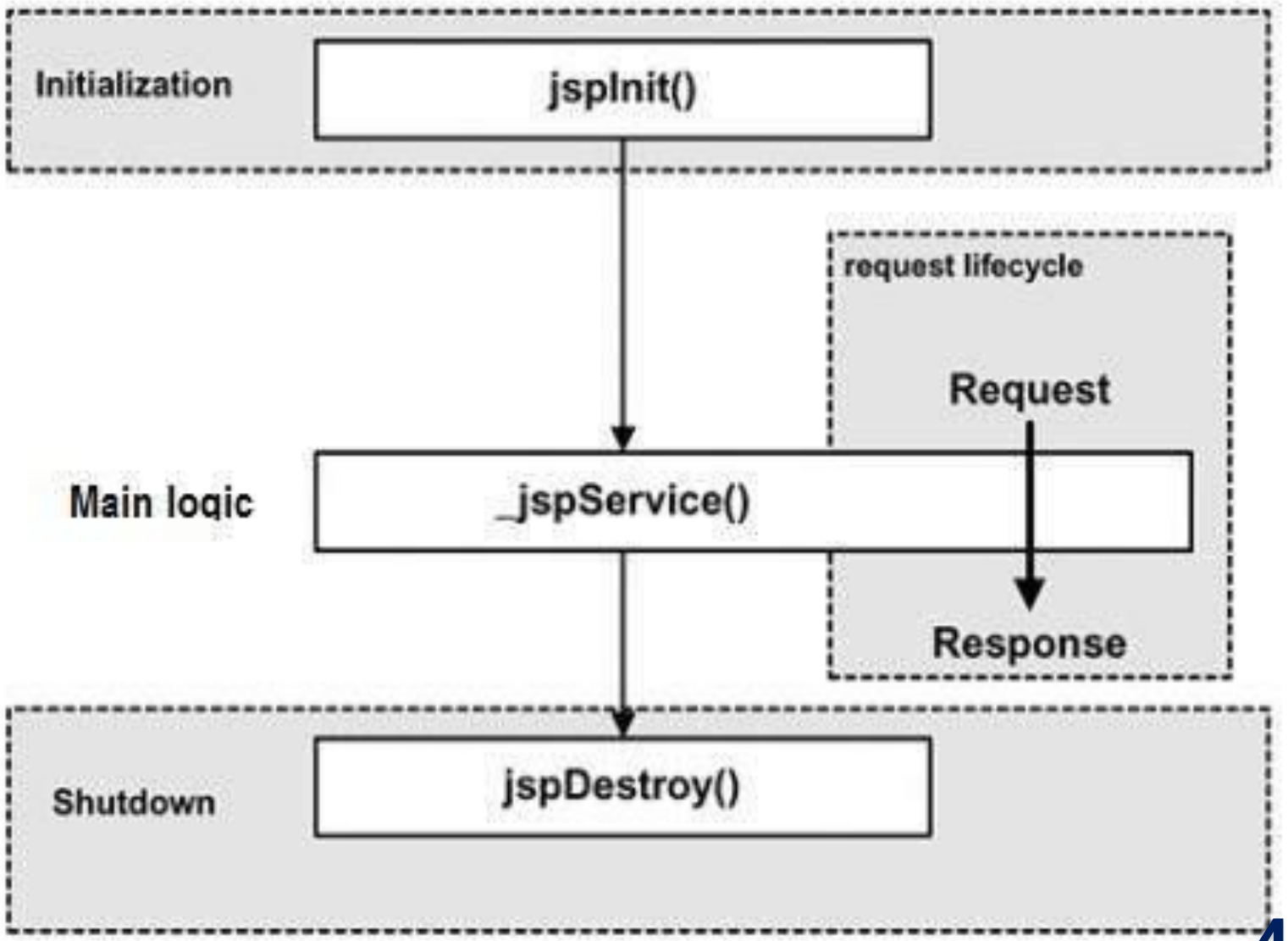
4. Then this JSP page is converted into corresponding servlet. This phase is also called as **translation phase**. The output of a translation phase is a Servlet.
5. The Servlet is then compiled to generate the servlet class file. Using this class file response is generate. This phase is called as **request processing phase**.
6. The JSP container will execute the servlet class.
7. A response page is return to the client



LIFE CYCLE OF JSP (cont.)

- The following are the paths followed by a JSP

- Compilation
- Initialization
- Execution
- Cleanup



LIFE CYCLE OF JSP (cont.)

JSP Compilation:

- When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page.

- If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

LIFE CYCLE OF JSP (cont.)

- The compilation process involves three steps:

- Parsing the JSP.
- Turning the JSP into a servlet.
- Compiling the servlet.

LIFE CYCLE OF JSP (cont.)

JSP Initialization:

- When a container loads a JSP it invokes the jspInit() method before servicing any requests. If you need to perform JSP-specific initialization, override the jspInit() method:

- public void jspInit(){
- // Initialization code...}

- Typically initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

LIFE CYCLE OF JSP (cont.)

JSP Execution:

- This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.
- Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the _jspService() method in the JSP.
- The _jspService() method takes an HttpServletRequest and an HttpServletResponse as its parameters as follows:
- void _jspService(HttpServletRequest request, HttpServletResponse response)
- { // Service handling code...}

LIFE CYCLE OF JSP (cont.)

JSP Cleanup:

- The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.
- The jspDestroy() method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.
- public void jspDestroy()
- { // Your cleanup code goes here. }

JSP IMPLICIT OBJECTS

JSP Implicit Objects

- There are **9 jsp implicit objects**. These objects are *created by the web container* that are available to all the jsp pages.
- The available implicit objects are out, request, config, session, application etc.
- The implicit objects are **pre-defined variable** used to access request and application data.
- A list of the 9 implicit objects is given in next slide.

JSP Implicit Objects (Cont.)

Object	Class/Interface	Meaning
out	JspWriter	It provides method related to I/O
request	HttpServletRequest	It provides method for accessing information made by current request
response	HttpServletResponse	It provides method for sending information
config	ServletConfig	For creating config object
application	ServletContext	For creating context object
session	HttpSession	This variable is used for accessing current client session
exception	Throwable	Used for handling Errors
page	Object	Used for handling page as an object
pageContext	PageContext	To access attributes from one of the four scopes.

JSP Implicit Object – out

- For writing any data to the buffer, JSP provides an implicit object named out. It is the object of JspWriter. In case of servlet you need to write:

```
PrintWriter out=response.getWriter();
```

- Example:

```
<html>
<body>
<% out.print("Today is:"+ new java.util.Date()); %>
</body>
</html>
```

JSP Implicit Object - request

- **JSP request** is an implicit object of type HttpServletRequest i.e. created for each jsp request by the web container.
- It can be used to get request information such as parameter, header information, remote address, server name, server port, content type, character encoding etc.
- It can also be used to set, get and remove attributes from the jsp request scope.

JSP Implicit Object – request – Example

- **index.html**

```
<form action="welcome.jsp">
<input type="text" name="uname"> <br/>
<input type="submit" value="go">
</form>
```

- **welcome.jsp**

```
<%
 String name=request.getParameter("uname");
 out.print("welcome "+name);
%>
```

JSP Implicit Object – response

- **JSP response** is an implicit object of type HttpServletResponse. The instance of HttpServletResponse is created by the web container for each jsp request.
- It can be used to add or manipulate response such as redirect response to another resource, send error etc.

JSP Implicit Object – response – Example

- **index.html**

```
<form action="welcome.jsp">
<input type="text" name="uname"> <br/>
<input type="submit" value="go">
</form>
```

- **welcome.jsp**

```
<%
response.sendRedirect("http://www.google.com");
%>
```

JSP Implicit Object – config

- **JSP config** is an implicit object of type *ServletConfig*.
- This object can be used to get initialization parameter for a particular JSP page.
- The config object is created by the web container for each jsp page.
- Generally, it is used to get initialization parameter from the web.xml file.

Example of config implicit object:

- Three files required:
  - index.html
  - web.xml file
  - welcome.jsp

JSP Implicit Object – config – Example: index.html

```
<form action="welcome.jsp">
Name:   <input type="text" name="uname">
        <br/>
        <input type="submit" value="go">
</form>
```

JSP Implicit Object – config – Example: web.xml

```
<web-app>
<servlet>
    <servlet-name>Sample</servlet-name>
    <jsp-file>/welcome.jsp</jsp-file>
    <init-param>
        <param-name>dname</param-name>
        <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>Sample</servlet-name>
    <url-pattern>/welcome</url-pattern>
</servlet-mapping>
</web-app>
```

JSP Implicit Object – config – Example: welcome.jsp

```
<%
out.print("Welcome "+request.getParameter("uname"));
String driver = config.getInitParameter("dname");
out.print("driver name is="+driver);
%>
```

JSP Implicit Object – application

- **JSP application** is an implicit object of type *ServletContext*.
- The instance of ServletContext is created only once by the web container when application or project is deployed on the server.
- This object can be used to get initialization parameter from configuaration file (web.xml).
- It can also be used to get, set or remove attribute from the application scope.
- This initialization parameter can be used by all jsp pages.

JSP Implicit Object – application – Example: index.html

```
<form action="welcome">
Name:   <input type="text" name="uname">
        <br/>
        <input type="submit" value="go">
</form>
```

JSP Implicit Object – application – Example: web.xml

```
<web-app>
<servlet>
    <servlet-name>Sample</servlet-name>
    <jsp-file>/welcome.jsp</jsp-file>
</servlet>
<servlet-mapping>
    <servlet-name>Sample</servlet-name>
    <url-pattern>/welcome</url-pattern>
</servlet-mapping>
<context-param>
    <param-name>dname</param-name>
    <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</context-param>
</web-app>
```

JSP Implicit Object – application – Example: welcome.jsp

```
<%
out.print("Welcome "+request.getParameter("uname"));
String driver=application.getInitParameter("dname");
out.print("driver name is="+driver);
%>
```

JSP Implicit Object – session

- **JSP session** is an implicit object of type HttpSession.
- It can be uses to set, get or remove attribute or to get session information.

Example of session implicit object:

- Three files required:
  - index.html
  - welcome.jsp
  - second.jsp

JSP Implicit Object – session – Example: index.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
</body>
</html>
```

JSP Implicit Object – session – Example: welcome.jsp

```
<html>
<body>
<%
    String name=request.getParameter("uname");
    out.print("Welcome "+name);
    session.setAttribute("user", name);
%>
<a href="second.jsp">Second JSP page</a>
</body>
</html>
```

JSP Implicit Object – session – Example: second.jsp

```
<html>
<body>
<%
 String name = (String)session.getAttribute("user");
 out.print("Hello "+name);
%>
</body>
</html>
```

JSP Implicit Object – page

- **JSP page** is an implicit object of type Object class.
- This object is assigned to the reference of auto generated servlet class. It is written as:
- `Object page=this;`
- For using this object it must be cast to Servlet type.
- For example:

```
<% (HttpServlet)page.log("message"); %>
```

- Since, it is of type Object it is less used because you can use this object directly in jsp.For example:

```
<% this.log("message"); %>
```

JSP Implicit Object – pageContext

- **JSP pageContext** is an implicit object of type PageContext class.
- The pageContext object can be used to set, get or remove attribute from one of the following scopes:
  - page
  - request
  - session
  - application
- In JSP, page scope is the default scope.
- **Example of pageContext implicit object:**
- Three files required:
  - index.html
  - welcome.jsp
  - second.jsp

JSP Implicit Object – pageContext – Example: index.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
</body>
</html>
```

JSP Implicit Object – pageContext – Example: welcome.jsp

```
<html>
<body>
<%
    String name=request.getParameter("uname");
    out.print("Welcome "+name);
    pageContext.setAttribute("user", name,
            PageContext.SESSION_SCOPE);
%>
<a href="second.jsp">Second JSP page</a>
</body>
</html>
```

JSP Implicit Object – pageContext – Example: second.jsp

```
<html>
<body>
<%
String name = (String)pageContext.getAttribute("user",
                                PageContext.SESSION_SCOPE);
out.print("Hello "+name);
%>
</body>
</html>
```

JSP Implicit Object – exception

- **JSP exception** is an implicit object of type java.lang.Throwable class.
- This object can be used to print the exception. But it can only be used in error pages.It is better to learn it after page directive.
- Example:

error.jsp

```
<%@ page isErrorPage="true" %>
<html>
<body>
 Sorry following exception occured: <%= exception %>
</body>
</html>
```

JSP DIRECTIVES

JSP Directives

- The jsp directives are messages that tells the web container how to translate a JSP page into the corresponding servlet.
- There are three types of directives:
  - page directive
  - include directive
  - taglib directive
- Syntax of JSP Directive
-
```
<%@ directive attribute="value" %>
```

JSP Directives (Cont.)

There are three types of directive tag

Directive	Description
<%@ page ... %>	Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
<%@ include ... %>	Includes a file during the translation phase.
<%@ taglib ... %>	Declares a tag library, containing custom actions, used in the page

Page Directive

- The page directive defines attributes that apply to an entire JSP page.

1. import
2. contentType
3. extends
4. info
5. buffer
6. isELIgnored
7. language
8. errorPage
9. isErrorPage
10. IsThreadSafe

Page Directive 1: import

- The import attribute is used to ***import class, interface or all the members of a package***. It is similar to import keyword in java class or interface.

```
importTag.jsp
<html>
<body>

<%@ page import="java.util.Date" %>
Today is: <%= new Date() %>

</body>
</html>
```

Page Directive 2: contentType

- The contentType attribute ***defines the MIME type*** of the HTTP response.
- The default value is “text/html”.

```
<html>
<body>

<%@ page contentType=text/html %>

</body>
 </html>
```

Page Directive 3: Buffer

- The buffer attribute ***sets the buffer size in kilobytes to handle output*** generated by the JSP page.
- The default size of the buffer is 8Kb.

```
<html>
<body>

<%@ page buffer="16kb" %>
Today is: <%= new java.util.Date() %>

</body>
</html>
```

Prog: Use of import, contentType and buffer Page Directives

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <title>Import,contentType and
          buffer tag Example</title>
    </head>
    <body>
        <%@page buffer="16kb" %>
        <%@ page import="java.util.Date" %>
            Today is: <%= new Date() %>
    </body>
</html>
```

Page Directive 4: isThreadSafe

1. Servlet and JSP both supports multithreaded.
2. If you want to control this behaviour of JSP page, you can use isThreadSafe attribute of page directive.
3. The value of isThreadSafe value is true.
4. If you make it false, the web container will serialise the multiple requests, i.e. it will wait until the JSP finishes responding to a request before passing another request to it.
5. Its syntax is:

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

Page Directive 5: info

This attribute simply **sets the information** of the JSP page.

```
<html>
<body>

<%@ page info="Made in India" %>
Today is: <%= new java.util.Date() %>

</body>
</html>
```

Page Directive 6: errorPage

The errorPage attribute is **used to define the error page**, if exception occurs in the current page, it will be **redirected** to the error page.

index.jsp

```
<html>
<body>
<%@ page errorPage="myerrorpage.jsp" %>
<%= 100/0 %>
</body>
</html>
```

Page Directive 7: isErrorPage

- The isErrorPage attribute is used **to declare that the current page is the error page.**

```
myerrorpage.jsp
<html>
<body>
<%@ page isErrorPage="true" %>
 Sorry an exception occured!<br/>
The exception is: <%= exception %>
</body>
</html>
```

Page Directive 8: isELIgnored

- We can ignore the Expression Language (EL) in jsp by the isELIgnored attribute.
- By default its value is false i.e. Expression Language is enabled by default.
- Syntax:

```
<%@ page isELIgnored="true" %>//Now EL will be ignored
```

Page Directive 9: language

- The language attribute **specifies the scripting language** used in the JSP page. The default value is "java".

Page Directive 10: extends

- The extends attribute defines the **parent class that will be inherited** by the generated servlet. It is rarely used.

JSP Include Directive

- The include directive is **used to include the contents** of any resource it may be **jsp file, html file or text file**.
- Syntax of include directive
- `<%@ include file="resourceName" %>`

Prog: JSP Include Directive

- In this example, we are including the content of the index.html file in a include_action.jsp page

```
<html>
<body>
<%@ include file="index.html" %>
</body>
</html>
```

JSP Taglib directive

- The JavaServer Pages API allows you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior.
- The custom tags are those that are created by user.
- The taglib directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides a means for identifying the custom tags in your JSP page.
- The taglib directive follows the following syntax:
- Syntax JSP Taglib directive:

```
<%@ taglib uri="uriofthetaglibrary"
          prefix="prefixoftaglibrary" %>
```

JSP Taglib directive (Cont.)

- Where the uri attribute value resolves to a location the container understands and the prefix attribute informs a container what bits of markup are custom actions.

- You can write XML equivalent of the above syntax as follows:

```
<jsp:directive.taglib uri="uri" prefix="prefixOfTag" />
```

JSP ACTION TAGS

JSP ACTION TAGs

- The action tags are used to **control the flow between pages** and to use Java Bean.
- The JSP action tags are given below:

JSP Action Tags	Description
jsp:forward	forwards the request and response to another resource.
jsp:include	includes another resource.
jsp:useBean	creates or locates bean object.
jsp:setProperty	sets the value of property in bean object.
jsp:getProperty	prints the value of property of the bean.
jsp:plugin	embeds another components such as applet.
jsp:param	sets the parameter value. It is used in forward and include mostly.

FORWARD ACTION AND PARAM TAG

- The jsp:forward action tag is used to ***forward the request to another resource it may be jsp, html or another resource***.

Syntax of jsp:forward action tag:

```
<jsp:forward page="relativeURL | <%= expression %>" />
```

Syntax of jsp:forward action tag with parameter:

```
<jsp:forward page="relativeURL | <%= expression %>">
<jsp:param name="parametername" value="parametervalue |
                <%=expression%>" />
</jsp:forward>
```

Prog: Forward request to other page using forward action:

Files need to be created:

- index.jsp
- second.html

Prog: Forward request to other page using forward action

- File: index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<body>
<h1>This page contains forward</h1>
<jsp:forward page="second.html"/>
<!-- Will forward request to the next page -->
</body>
</html>
```

Prog: Forward request to other page using forward action

- File: second.html

```
<html>
<body>
<h1>You are in a forwarded page</h1>
</body>
</html>
```

jsp:include action tag

- The **jsp:include action tag** is used to include the content of another resource it may be jsp, html or servlet.
- The jsp include action tag includes the resource at request time so it is **better for dynamic pages** because there might be changes in future.
- The jsp:include tag can be used to include static as well as dynamic pages.
- Using the <jsp:include> action we can **include different files in current jsp page.**
- Advantage of jsp:include action tag:
  - **Code reusability** : We can use a page many times such as including header and footer pages in all pages. So it saves a lot of time.

Prog: To include different types of files in a jsp page

- File: index.html

```
<html>
  <body>
          <h3>This is a html page</h3>
     </body>
</html>
```

Prog: To include different types of files in a jsp page

- File: newjsp.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<body>
<%--returns hour--%>
<br> Time is <%= new java.util.Date().getHours() %>
<%--returns min--%>
: <%= new java.util.Date().getMinutes()%>
<%--returns sec--%>
: <%= new java.util.Date().getSeconds()%>
</body>
</html>
```

Prog: To include different types of files in a jsp page

- File: includejsp.jsp

```
<%@page contentType="text/html"
                    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<body>
   <h1>This page contains include action</h1>
   <jsp:include page="index.html"/>
   <jsp:include page="newjsp.jsp"/>
</body>
</html>
```

jsp:useBean action tag

- The jsp:useBean action tag is used to **locate or instantiate a bean class.**
- If bean object of the Bean class is already created, it doesn't create the bean depending on the scope.
- But if object of bean is not created, it instantiates the bean.
- **Syntax of jsp:useBean action tag**

<jsp:useBean id= "instanceName" scope= "page | request | session |

application" **class**= "packageName.className"

type= "packageName.className" beanName="packageName.className | <%= expression >" >

</jsp:useBean>

Syntax of jsp:useBean action tag

```
<jsp:useBean id= "instanceName" scope= "page | request | session | application"   class= "packageName.className" " >
</jsp:useBean>
```

- **id**: is used to identify the bean in the specified scope.
- **scope**: represents the scope of the bean. It may be page, request, session or application. The default scope is page.
- **page**: specifies that you can use this bean within the JSP page. The default scope is page.
- **request**: specifies that you can use this bean from any JSP page that processes the same request. It has wider scope than page.
- **session**: specifies that you can use this bean from any JSP page in the same session whether processes the same request or not. It has wider scope than request.
- **application**: specifies that you can use this bean from any JSP page in the same application. It has wider scope than session.
- **class**: instantiates the specified bean class (i.e. creates an object of the bean class)

jsp:setProperty and jsp:getProperty action tags

- The setProperty and getProperty action tags are used for developing web application with Java Bean.
- In web development, bean class is mostly used because it is a reusable software component that represents data.
- The jsp:setProperty action tag sets a property value or values in a bean using the setter method.
- **Syntax of jsp:setProperty action tag**

```
<jsp:setProperty name="instanceOfBean" property= "*"
  |  property="propertyName" param="parameterName"
  |  property="propertyName" value="{ string |
                    <%= expression %>}" />
```

index.html

```
<form action="process.jsp" method="post">
Name:<input type="text" name="name"><br>
Password:<input type="password" name="password"><br>
Email:<input type="text" name="email"><br>
<input type="submit" value="register">
</form>
```

process.jsp

```
<jsp:useBean id="u" class="org.sssit.User"></jsp:useBean>
<jsp:setProperty property="*" name="u"/>

Record:<br>
<jsp:getProperty property="name" name="u"/><br>
<jsp:getProperty property="password" name="u"/><br>
<jsp:getProperty property="email" name="u" /><br>
```

User.java

```
package org.sssit;

public class User{
private String name,password,email;

    public void setName(String name){
        this.name=name;
    }
    public String getName(){
        return name;
    }

    public void setPassword(String Password){
        this.password=password;
    }
    public String getPassword(){
        return password;
    }

    public void setEmail(String Email){
        this.email=email;
    }
    public String getEmail(){
        return email;
    }

}
```

Expression Language (EL) in JSP

Expression Language (EL) in JSP

- The **Expression Language** (EL) simplifies the accessibility of **data stored** in the **Java Bean component**, and other objects like **request, session, application etc**.
- There are many implicit objects, operators and reserve words in EL.
- Syntax for Expression Language (EL)

```
${ expression }
```

Expression Language (EL) in JSP (Cont.)

Variables

- The web container evaluates a variable that appears in an expression by looking up its value according to the behavior of PageContext.findAttribute(String).
- For example, when evaluating the expression ${product}, the container will look for product in the page, request, session, and application scopes and will return its value.
- If product is not found, null is returned. A variable that matches one of the implicit objects described in Implicit Objects will return that implicit object instead of the variable's value.

Expression Language (EL) in JSP (Cont.)

- Implicit Objects

- The JSP expression language defines a set of implicit objects:
- pageContext: The context for the JSP page. Provides access to various objects including:
- servletContext: The context for the JSP page's servlet and any web components contained in the same application. See Accessing the Web Context.
- session: The session object for the client. See Maintaining Client State.
- request: The request the execution of the JSP page. See Getting Information from Requests.
- response: The response returned by the JSP page. See Constructing Responses.

Expression Language (EL) in JSP (Cont.)

- In addition, several implicit objects are available that allow easy access to the following objects:

- param: Maps a request parameter name to a single value
- paramValues: Maps a request parameter name to an array of values
- header: Maps a request header name to a single value
- headerValues: Maps a request header name to an array of values
- cookie: Maps a cookie name to a single cookie
- initParam: Maps a context initialization parameter name to a single value

Expression Language (EL) in JSP (Cont.)

Literals

- The JSP expression language defines the following literals:

- Boolean: true and false
- Integer: as in Java
- Floating point: as in Java
- String: with single and double quotes; " is escaped as \", ' is escaped as \', and \ is escaped as \\.
- Null: null

Expression Language (EL) in JSP (Cont.)

- Operators

- In addition to the . and [] operators discussed in Variables, the JSP expression language provides the following operators:

- Arithmetic: +, - (binary), *, / and div, % and mod, - (unary)
- Logical: and, &&, or, ||, not, !
- Relational: ==, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le. Comparisons can be made against other values, or against boolean, string, integer, or floating point literals.
- Empty: The empty operator is a prefix operation that can be used to determine whether a value is null or empty.
- Conditional: A ? B : C. Evaluate B or C, depending on the result of the evaluation of A.

Expression Language (EL) in JSP (Cont.)

- Reserved Words

- The following words are reserved for the JSP expression language and should not be used as identifiers.

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

EL param example

In this example, we have created two files index.jsp and process.jsp.
The index.jsp file gets input from the user and sends the request to the process.jsp which in turn prints the name of the user using EL.

index.jsp

```
<form action="process.jsp">
Enter Name:<input type="text" name="name" /><br/><br/>
<input type="submit" value="go"/>
</form>
```

process.jsp

```
Welcome, ${ param.name }
```

EL sessionScope example

In this example, we printing the data stored in the session scope using EL. For this purpose, we have used sessionScope object.

index.jsp

```
<h3>welcome to index page</h3>
<%
session.setAttribute("user","umangya..!!");
%>
<a href="process.jsp">visit</a>
```

process.jsp

```
Value is ${ sessionScope.user }
```

EL cookie example

index.jsp

```
<h1>First JSP</h1>
<%
Cookie ck=new Cookie("name","umangya..!");
response.addCookie(ck);
%>
<a href="process.jsp">click</a>
```

process.jsp

```
Hello, ${cookie.name.value}
```

JSP
STANDARD TAG LIBRARIES
(JSTL)

JSP - Standard Tag Library (JSTL)

- The JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates core functionality common to many JSP applications.

- JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags.

- It also provides a framework for integrating existing custom tags with JSTL tags.

JSP STANDARD TAG LIBRARIES (JSTL)

- The JSP Standard Tag Library (JSTL) represents a set of tags to simplify the JSP development.
- **Advantage of JSTL:**

1. **Fast Developement:** JSTL provides many tags that simplifies the JSP.
2. **Code Reusability:** We can use the JSTL tags in various pages.
3. **No need to use scriptlet tag:** It avoids the use of scriptlet tag.

- The JSTL tags can be classified, according to their functions, into following JSTL tag library groups that can be used when creating a JSP page:
- **Core Tags, Formatting tags, Function tags**
- **SQL tags, XML tags**

JSTL Tags:

Tag Name	Description
Core tags	The JSTL core tag provide variable support, URL management, flow control etc. The prefix of core tag is c.
Function tags	The functions tags provide support for string related function. The prefix is fn.
SQL tags	The JSTL sql tags provide SQL support. The prefix is sql.

CORE TAG:

- The JSTL core tag provides **variable support, URL management, flow control etc**. The syntax used for including JSTL core library in your JSP is:
  - <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

Tags	Description
c:out	It display the result of an expression, similar to the way <%=...%> tag work.
c:import	It Retrives relative or an absolute URL and display the contents.
c:set	It sets the result of an expression under evaluation in a 'scope' variable.
c:remove	It is used for removing the specified scoped variable from a particular scope.
c:catch	It is used for Catches any Throwable exceptions that occurs in the body.
c:if	It is conditional tag used for testing the condition and display the body content only if the expression evaluates is true.
c:choose, c:when, c:otherwise	It is the simple conditional tag that includes its body content if the evaluated condition is true.
c:forEach	It is the basic iteration tag. It repeats the nested body content for fixed number of times or over collection.
c:forTokens	It iterates over tokens which is separated by the supplied delimeters.
c:param	It adds a parameter in a containing 'import' tag's URL.
c:redirect	It redirects the browser to a new URL and supports the context-relative URLs.

JSTL Core <c:out> Tag

- The **<c:out> tag is similar to JSP expression tag**. It will display the result of an expression, similar to the way < %=...% > work.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefi
x="c" %>
     <html>
          <head>
          <title>Tag Example</title>
          </head>
          <body>
               <c:out value="${'Welcome to JSTL'}"/>
          </body>
     </html>
```

JSTL Core <c:set> Tag

- It is **used to set the result of an expression evaluated in a 'scope'.** The <c:set> tag is helpful because it evaluates the expression and use the result to set a value of JavaBean.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefi
x="c" %>
<html>
<head>
<title>Core Tag Example</title>
</head>
<body>
<c:set var="Income" scope="session" value="${4000*4}"/>
<c:out value="${Income}"/>
</body>
</html>
```

JSTL Core <c:remove> Tag

It is used for **removing the specified variable from a particular scope**.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
        <html>
        <head>
        <title>Core Tag Example</title>
        </head>
        <body>
        <c:set var="income" scope="session" value="${4000*4}"/>
        <p>Before Remove Value is: <c:out value="${income}"/></p>
        <c:remove var="income"/>
        <p>After Remove Value is: <c:out value="${income}"/></p>
        </body>
        </html>
```

JSTL Core <c:if> Tag

The **< c:if > tag is used for testing the condition** and it display the body content, if the expression evaluated is true.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Core Tag Example</title>
</head>
<body>
<c:set var="income" scope="session" value="${4000*4}"/>
<c:if test="${income > 8000}">
  <p>My income is: <c:out value="${income}"/><p>
</c:if>
</body>
</html>
```

JSTL Core <c:catch> Tag

It is used for **Catches any Throwable exceptions** that occurs in the body and optionally exposes it. In general **it is used for error handling** and to deal more easily with the problem occur in program.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Core Tag Example</title>
</head>
<body>

<c:catch var ="catchtheException">
   <% int x = 2/0;%>
</c:catch>

<c:if test = "${catchtheException != null}">
   <p>The type of exception is : ${catchtheException} </p>
</c:if>

</body>
</html>
```

JSTL Core <c:choose>, <c:when>, <c:otherwise> Tag

The **<c:choose> tag** is a conditional tag that establish a context for mutually exclusive conditional operations. It works like a Java **switch** statement in which we choose between a numbers of alternatives.

The **<c:when > is subtag of <choose >** that will include its body if the condition evaluated be '**true**'.

The **< c:otherwise > is also subtag of < choose >** it follows & <when > tags and runs only if all the prior condition evaluated is '**false**'.

The c:when and c:otherwise works like **if-else statement**. But it must be placed inside c:choose tag.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Core Tag Example</title>
</head>
<body>
<c:set var="income" scope="session" value="${4000*4}"/>
<p>Your income is : <c:out value="${income}"/></p>
<c:choose>
    <c:when test="${income <= 1000}">
       Income is not good.
    </c:when>
    <c:when test="${income > 10000}">
        Income is very good.
    </c:when>
    <c:otherwise>
       Income is undetermined...
    </c:otherwise>
</c:choose>
</body>
</html>
```

JSTL Core <c:forEach> Tag

The **<c:for each > is an iteration tag** used for repeating the nested body content for fixed number of times or over the collection.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Core Tag Example</title>
</head>
<body>
<c:forEach var="j" begin="1" end="3">
   Item <c:out value="${j}"/><p>
</c:forEach>
</body>
</html>
```

Output:

Item 1

Item 2

Item 3

JSTL Core <c:redirect> Tag

The **< c:redirect > tag redirects the browser to a new URL.**

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Core Tag Example</title>
</head>
<body>
  <c:set var="url" value="0" scope="request"/>
  <c:if test="${url<1}">
     <c:redirect url="http://javatpoint.com"/>
  </c:if>
  <c:if test="${url>1}">
     <c:redirect url="http://facebook.com"/>
  </c:if>
</body>
</html>
```

FUNCTION TAG LIBRARY

The JSTL function provides a number of standard functions, **most of these functions are common string manipulation functions.** The syntax used for including JSTL function library in your JSP is:

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

JSTL Functions	Description
fn:contains()	It is used to test if an input string containing the specified substring in a program.
fn:containsIgnoreCase()	It is used to test if an input string contains the specified substring as a case insensitive way.
fn:endsWith()	It is used to test if an input string ends with the specified suffix.
fn:indexOf()	It returns an index within a string of first occurrence of a specified substring.
fn:trim()	It removes the blank spaces from both the ends of a string.
fn:startsWith()	It is used for checking whether the given string is started with a particular string value.
fn:split()	It splits the string into an array of substrings.
fn:toLowerCase()	It converts all the characters of a string to lower case.
fn:toUpperCase()	It converts all the characters of a string to upper case.
fn:substring()	It returns the subset of a string according to the given start and end position.
fn:substringAfter()	It returns the subset of string after a specific substring.
fn:substringBefore()	It returns the subset of string before a specific substring.
fn:length()	It returns the number of characters inside a string, or the number of items in a collection.
fn:replace()	It replaces all the occurrence of a string with another string sequence.

Conversion of given string to lowercase

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<html>
<head>
<title>Using JSTL Function </title>
</head>
<body>
  <c:set var="str1" value="I LOVE INDIA"/>
  <c:set var="str2" value="${fn:toLowerCase(str1)}"/>
  <p><b>Original :</b>${str1}</p>
  <p><b>Conversion :</b>${str2}</p>
</body>
</html>
```

OUT PUT:

Original : I LOVE INDIA

Conversion : i love india

SQL TAG LIBRARY

The JSTL sql tags provide SQL support. The url for the sql tags is **http://java.sun.com/jsp/jstl/sql** and prefix is **sql**.

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

SQL Tags	Descriptions
sql:setDataSource	It is used for creating a simple data source
sql:query	It is used for executing the SQL query defined in its sql attribute or the body.
sql:update	It is used for executing the SQL update defined in its sql attribute or in the tag body.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>
<html>
<head>
<title>sql:setDataSource Tag</title>
</head>
<body>

<sql:setDataSource var="db" driver="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost/test"
    user="root"  password="1234"/>

<sql:query dataSource="${db}" var="rs">
SELECT * from Students;
</sql:query>

<table border="2" width="100%">
<tr>
<th>Student ID</th>
<th>First Name</th>
<th>Last Name</th>
<th>Age</th>
</tr>
```

```
<c:forEach var="table" items="${rs.rows}">
<tr>
<td><c:out value="${table.id}"/></td>
<td><c:out value="${table.First_Name}"/></td>
<td><c:out value="${table.Last_Name}"/></td>
<td><c:out value="${table.Age}"/></td>
</tr>

</c:forEach>
</table>
</body>
</html>
```

FORMATTING TAG LIBRARY

- Formatting tags:
- The JSTL formatting tags are used to format and display text, the date, the time, and numbers for internationalized Web sites. Following is the syntax to include Formatting library in your JSP:

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

FORMATTING TAG LIBRARY

Tag	Description
<fmt:formatNumber>	To render numerical value with specific precision or format.
<fmt:parseNumber>	Parses the string representation of a number, currency, or percentage.
<fmt:formatDate>	Formats a date and/or time using the supplied styles and pattern
<fmt:parseDate>	Parses the string representation of a date and/or time
<fmt:bundle>	Loads a resource bundle to be used by its tag body.
<fmt:setLocale>	Stores the given locale in the locale configuration variable.
<fmt:setBundle>	Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable.
<fmt:timeZone>	Specifies the time zone for any time formatting or parsing actions nested in its body.
<fmt:setTimeZone>	Stores the given time zone in the time zone configuration variable
<fmt:message>	To display an internationalized message.
<fmt:requestEncoding>	Sets the request character encoding

JSP XML tags

- The **XML tags are useful for creating and manipulating the XML documents through the JSP.**
- The **url** for the xml tags is **http://java.sun.com/jsp/jstl/xml** and **prefix is x.**
- The syntax used for including JSTL XML tags library in your JSP is:

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

XML Tags	Descriptions
x:out	Similar to <%= ... > tag.
x:parse	It is used for parse the XML data specified either in the tag body or an attribute.
x:set	It is used to sets a variable to the value of an XPath expression.
x:choose	It is a conditional tag that establish a context for mutually exclusive conditional operations.
x:when	It is a subtag of that will include its body if the condition evaluated be 'true'.
x:otherwise	It is subtag of that follows tags and runs only if all the prior conditions evaluated be 'false'.
x:if	It is used for evaluating the test XPath expression and if it is true, it will processes its body content.
x:transform	It is used in a XML document for providing the XSL(Extensible Stylesheet Language) transformation.
x:param	It is used along with the transform tag for setting the parameter in the XSLT style sheet.

JSTL CUSTOM TAGS

JSP Tag Extensions

- Tag extensions or custom tags is a user-defined JSP language element. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler.

- The Web container then invokes those operations when the JSP page's servlet is executed.

- JSP tag extensions let you create new tags that you can insert directly into a JavaServer.

Create "Hello" Tag:

- Consider you want to define a custom tag named <ex:Hello> and you want to use it in the following fashion without a body:

- <ex:Hello />

- The next code has simple coding where doTag() method takes the current JspContext object using getJspContext() method and uses it to send "Hello Custom Tag!" to the current JspWriter object.

Create "Hello" Tag (Cont.)

- To create a custom JSP tag, you must first create a Java class that acts as a tag handler. So let us create HelloTag class as follows:

```
package com.tutorialspoint;
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;
public class HelloTag extends SimpleTagSupport {
  public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    out.println("Hello Custom Tag!");
  }
}
```

Create "Hello" Tag (Cont.)

- Now it's time to use above defined custom tag Hello in our JSP program as follows:

- <%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
- <html>
- <head>
- <title>A sample custom tag</title>
- </head>
- <body>
- <ex:Hello/>
- </body>
- </html>

Custom Tag Attributes:

Property	Purpose
name	The name element defines the name of an attribute. Each attribute name must be unique for a particular tag.
required	This specifies if this attribute is required or optional. It would be false for optional.
rtexprvalue	Declares if a runtime expression value for a tag attribute is valid
type	Defines the Java class-type of this attribute. By default it is assumed as **String**
description	Informational description can be provided.

Tag Handler API

- The JSP API defines a set of classes and interfaces that you use to write custom tag handlers.
- Your tag handler must be of one of the following two types:

- Classic Tag Handlers implement one of three interfaces:

1. Tag

- Implement the javax.servlet.jsp.tagext.Tag interface .The API also provides a convenience class TagSupport that implements the Tag interface and provides default empty methods for the methods defined in the interface.

Tag Handler API (Cont.)

2. BodyTag

- Implement the javax.servlet.jsp.tagext.BodyTag interface if your custom tag needs to use a body. The API also provides a convenience class BodyTagSupport that implements the BodyTag interface and provides default empty methods for the methods defined in the interface.

3. IterationTag

- Implement the javax.servlet.jsp.tagext.IterationTag interface to extend Tag by defining an additional method doAfterBody() that controls the reevaluation of the body.

Tag Handler API (Cont.)

Simple Tag Handlers (SimpleTag interface):

- Implement the javax.servlet.jsp.tagext.SimpleTag interface if you wish to use a much simpler invocation protocol.

- The SimpleTag interface does not extend the javax.servlet.jsp.tagext.Tag interface as does the BodyTag interface.

- Therefore, instead of supporting the doStartTag() and doEndTag() methods, the SimpleTag interface provides a simple doTag() method, which is called once and only once for each tag invocation.

Tag Handler

- A tag handler has access to an API that allows it to communicate with the JSP page.
- The entry points to the API are two objects: the JSP context (javax.servlet.jsp.JspContext) for simple tag handlers and the page context (javax.servlet.jsp.PageContext) for classic tag handlers.
- A tag handler can retrieve all the other implicit objects (request, session, and application) that are accessible from a JSP page through these objects.
- In addition, implicit objects can have named attributes associated with them. Such attributes are accessed using [set|get]Attribute methods.

Tag Handler (Cont.)

- Tag handler methods defined by the Tag and BodyTag interfaces are called by the JSP page's servlet at various points during the evaluation of the tag.

- To provide a tag handler implementation, you must implement the methods, summarized in Table that are invoked at various stages of processing the tag.

Tag Handler Methods

Tag Type	Interface	Methods
Basic	Tag	doStartTag, doEndTag
Attributes	Tag	doStartTag, doEndTag, setAttribute1,…, N, release
Body	Tag	doStartTag, doEndTag, release
Body, iterative evaluation	IterationTag	doStartTag, doAfterBody, doEndTag, release
Body, manipulation	BodyTag	doStartTag, doEndTag, release, doInitBody, doAfterBody

JspFragment

- Encapsulates a portion of JSP code in an object that can be invoked as many times as needed.

- The definition of the JSP fragment must only contain template text and JSP action elements. In other words, it must not contain scriptlets or scriptlet expressions.

- At translation time, the container generates an implementation of the JspFragment abstract class capable of executing the defined fragment.

Tag files

- A tag file is a source file that contains a fragment of JSP code that is reusable as a custom tag.

- Tag files allow you to create custom tags using JSP syntax. Just as a JSP page gets translated into a servlet class and then compiled, a tag file gets translated into a tag handler and then compiled.

- The recommended file extension for a tag file is .tag.

Tag File Directives

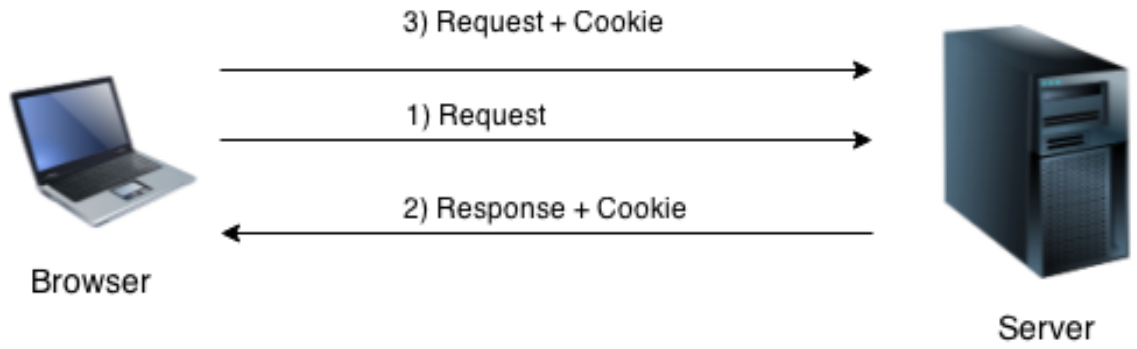
- Directives are used to control aspects of tag file translation to a tag handler, and to specify aspects of the tag, attributes of the tag, and variables exposed by the tag.

Directive	Description
taglib	Identical to taglib for JSP pages.
include	Identical to include directive for JSP pages. Note that if the included file contains syntax unsuitable for tag files, a translation error will occur.
tag	Similar to the page directive in a JSP page, but applies to tag files instead of JSP pages. Also used for declaring custom tag properties such as display name
attribute	Declares an attribute of the custom tag defined in the tag file.
variable	Declares an EL variable exposed by the tag to the calling page.

JSP SESSION MANAGEMENT: COOKIES

JSP COOKIES HANDLING

1. By default, **each request is considered as a new request.**
2. In cookies technique, we add cookie with response from the Server.
3. So cookie is stored in the cache of the browser.
4. After that if request is sent by the user, cookie is added with request by default.
5. Thus, we recognize the user as the old user.



JSP COOKIES HANDLING

6. Cookies are small text file that are stored in the client computer.
7. These are basically used to keep track of user who browse the web.
8. The info stored in the cookie are generally name, age , id, city and so on.
9. The servlet container sends a set of cookies to the web browser.
10. The browser stores the cookies on the local machine and makes use of this information next time when the browser is browsing the web.
11. Cookies are usually set in HTTP header.

JSP COOKIES HANDLING

- **Advantage of Cookies:**

1. Simplest technique of maintaining the state.
2. Cookies are maintained at client side.

- **Disadvantage of Cookies:**

1. It will not work if cookie is disabled from the browser.

Prog: Files needed to create, delete and read cookies

1. index.html
2. createcookie.jsp
3. readcookie.jsp
4. deletecookie.jsp

PROG: IMPLEMENTATION OF COOKIES IN JSP

index.html

```
<html>
  <head>
    <title>TODO supply a title</title>
  </head>
  <body>
    <form action="createCookie.jsp" method="post">
      USERNAME:<input type="text" name="name"><br>
      CITY:<input type="text" name="city"><br>
      <input type="submit" value="submit">
    </form>
  </body>
</html>
```

createCookie.jsp

```
<body>
    <%
        String name = request.getParameter("name");
        String city = request.getParameter("city");

                                                    //creating cookie object
        Cookie nameCookie = new Cookie("name", name);
        Cookie cityCookie = new Cookie("city", city);
                                                    //creating cookie object

                                                    //adding cookie in the response
        response.addCookie(nameCookie);
        response.addCookie(cityCookie);
                                                    //adding cookie in the response

                                                    //changing the maximum age
        nameCookie.setMaxAge(60*60*24);
        cityCookie.setMaxAge(60*60*24);
                                                    //changing the maximum age
    %>
    <a href="readCookie.jsp" >Click here</a> to read the
cookies
  </body>
```

readCookie.jsp

```
<body>
        <h3>Reading the cookie</h3>
        <%
        Cookie [] cookies = request.getCookies();          //return all the cookies from the browser
        for(int i =0;i<cookies.length;i++){                //printing name and value of cookie

out.println("Cookie_name"+cookies[i].getName()+"<br>");

out.println("Cookie_value"+cookies[i].getValue()+"<br>");
        }
            %>
            <a href="deletecookies.jsp">Click here</a>To delete the
cookie..!!
    </body>
```

deleteCookie.jsp

```
<body>

        <%
        Cookie nameCookie = new Cookie("name", "");
        nameCookie.setMaxAge(0);
        nameCookie.setValue("");
        response.addCookie(nameCookie);

        Cookie cityCookie = new Cookie("city", "");
        cityCookie.setMaxAge(0);
        cityCookie.setValue("");
        response.addCookie(cityCookie);
        %>

        <a href="readCookie.jsp">Click here</a> to check the
deletion..!!
    </body>
```

EXCEPTION HANDLING

Exception Handling in JSP

- The exception is normally an object that is thrown at runtime.
- Exception Handling is the process to handle the runtime errors.
- There may occur exception any time in your web application.
- So handling exceptions is a safer side for the web developer.
- In JSP, there are two ways to perform exception handling:
  - By **errorPage** and **isErrorPage** attributes of page directive
  - By **<error-page>** element in web.xml file

exception handling in jsp by the elements of page directive

- In this case, you must define and create a page to handle the exceptions, as in the **error.jsp** page. The pages where may occur exception, define the errorPage attribute of page directive, as in the **process.jsp** page.

error.jsp

```
<%@ page isErrorPage="true" %>
<h3>Sorry an exception occured!</h3>
Exception is: <%= exception %>
```

error-page element in web.xml file

- This approach is better because you don't need to specify the errorPage attribute in each jsp page.

- Specifying the single entry in the web.xml file will handle the exception. In this case, either specify exception-type or error-code with the location element.

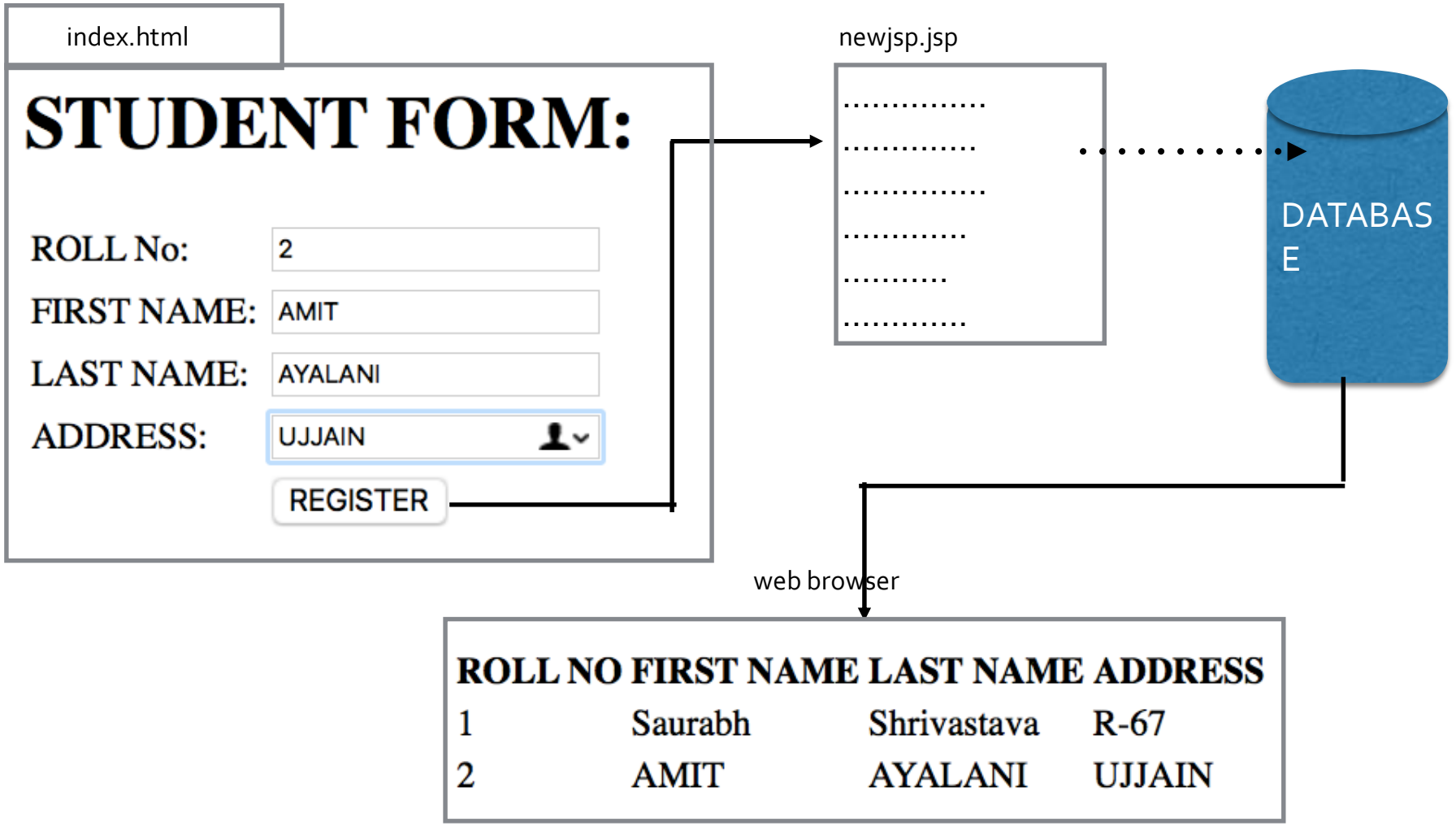
- If you want to handle all the exception, you will have to specify the java.lang.Exception in the exception-type element.

Cont.

```
<web-app>


 <error-page>
  <location>/error.jsp</location>
  </error-page>


</web-app>
```

JSP DATABASE ACCESS: CRUD APPLICATION

PROG: JSP DATABASE ACCESS



FILES NEED TO BE CREATED:

- index.html
- newjsp_1.jsp

index.html

```
<h1>STUDENT FORM:</h1>
    <form action="newjsp_1.jsp" method="GET">
    <table>
      <tr>
        <td>ROLL No:</td><td><input type="text" name="rollno" ></td>
      </tr>
      <tr>
        <td>FIRST NAME:</td><td><input type="text" name="fname" ></td>
      </tr>
      <tr>
        <td>LAST NAME:</td><td><input type="text" name="lname" ></td>
      </tr>
      <tr>
        <td>ADDRESS:</td><td><input type="text" name="addr" ></td>
      </tr>
      <tr>
        <td></td><td><input type="submit" value="REGISTER" ></td>
      </tr>

    </table>
     </form>
```

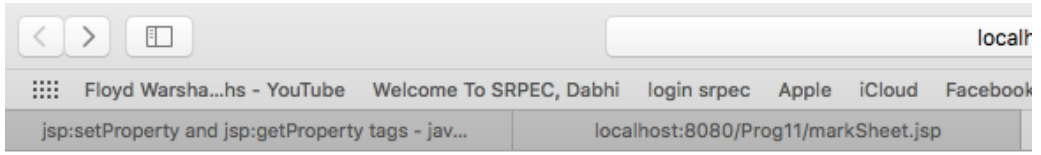
newjsp_1.jsp

```
<%@page language="java" import="java.sql.*" %>
<%@page import="java.io.*" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html>
  <body>
    <table>
      <tr>
        <td><b>ROLL NO</b></td>
        <td><b>FIRST NAME</b></td>
        <td><b>LAST NAME</b></td>
        <td><b>ADDRESS</b></td>
      </tr>
```

```
<%
        //read request parameter
        String rollno = request.getParameter("rollno");
        String fname = request.getParameter("fname");
        String lname = request.getParameter("lname");
        String addr = request.getParameter("addr");

            //initialize variable of driver

        String
driverClassName="com.mysql.jdbc.Driver";
        String url="jdbc:mysql://localhost:3306/ss";
        String user="root";
        String pwd="";

        //register jdbc driver or load driver

Class.forName(driverClassName).newInstance();

  //Open connection

        Connection
con=DriverManager.getConnection(url,user,pwd);

        //making a prepared statement

        PreparedStatement ps =
con.prepareStatement("insert into formtb
values(?,?,?,?)");

        ps.setString(1, rollno);

        ps.setString(2, fname);

        ps.setString(3, lname);

        ps.setString(4, addr);

        int p = ps.executeUpdate();

        PreparedStatement st =
con.prepareStatement("select * from formtb");

        ResultSet rs= st.executeQuery();

        while(rs.next()){

        %>
```

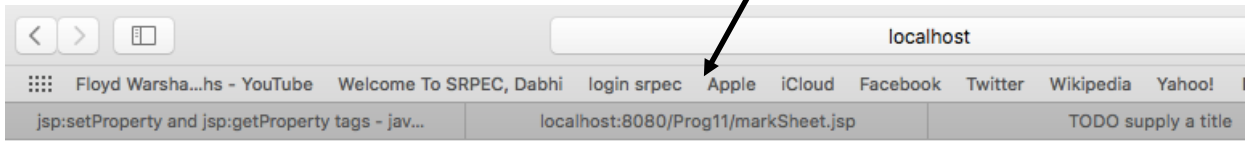
```
<tr>
        <td><%= rs.getString("rollno") %></td>
        <td><%= rs.getString("fname") %></td>
        <td><%= rs.getString("lname") %></td>
        <td><%= rs.getString("addr") %></td>
    </tr>
    <%
    }
        out.print("</table>");
        rs.close();
        ps.close();
        st.close();
        con.close();
    %>
  </table>
  </body>
</html>
```

PROG: To display semester mark sheet



STUDENT MARKSHEET

Enter Seat No: 111 Submit



MARK SHEET

SEAT NO	NAME	SUBJECT 1	SUBJECT 2	SUBJECT 3	STATUS
111	AAA	40	50	60	pass

index.html

```
<html>
   <head>
      <title>TODO supply a title</title>
      <meta charset="UTF-8">
      <meta name="viewport"
content="width=device-width, initial-scale=1.0">
   </head>
   <body>
      <h1>STUDENT MARKSHEET</h1>
      <form action="markSheet.jsp"
method="post">
         Enter Seat No: <input type="text"
name="seatno">
         <input type="submit" value="Submit">
      </form>
   </body>
</html>
```

markSheet.jsp

```
<%@page language="java" import="java.sql.*" %>
<%@page import="java.io.*" %>
<%@page contentType="text/html"
pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <body>
      <table>
         <tr>
            <td colspan="6"><center>MARK
SHEET</center></td>
         </tr>
         <tr>
            <td><b>SEAT NO</b></td>
            <td><b>NAME</b></td>
            <td><b>SUBJECT 1</b></td>
            <td><b>SUBJECT 2</b></td>
            <td><b>SUBJECT 3</b></td>
            <td><b>STATUS</b></td>
         </tr>

<%
         //read request parameter
         String seatno =
request.getParameter("seatno");

         //initialize variable of driver

         String
driverClassName="com.mysql.jdbc.Driver";
         String
url="jdbc:mysql://localhost:3306/marksheetDb";
         String user="root";
         String pwd="";

        //register jdbc driver or load driver
         Class.forName(driverClassName).newInstance();
         //Open connection
         Connection con=DriverManager.getConnection(url,user,pwd);

            //making a prepared statement and storing in resultset

         PreparedStatement st = con.prepareStatement("select * from
MarkList where "
               + "SeatNo ="+seatno);
         ResultSet rs= st.executeQuery();

         while(rs.next()){
         %>

            //reading from result set
          <tr>
            <td><%= rs.getString("SeatNo") %></td>
            <td><%= rs.getString("Name") %></td>
            <td><%= rs.getString("Subject1") %></td>
            <td><%= rs.getString("Subject2") %></td>
            <td><%= rs.getString("Subject3") %></td>
            <td><%= rs.getString("Status") %></td>
         </tr>

         <%
         }
            out.print("</table>");
            rs.close();
            st.close();
            con.close();
         %>
      </table>
      </body>
</html>
```

JSP APPLICATION DESIGN WITH MVC

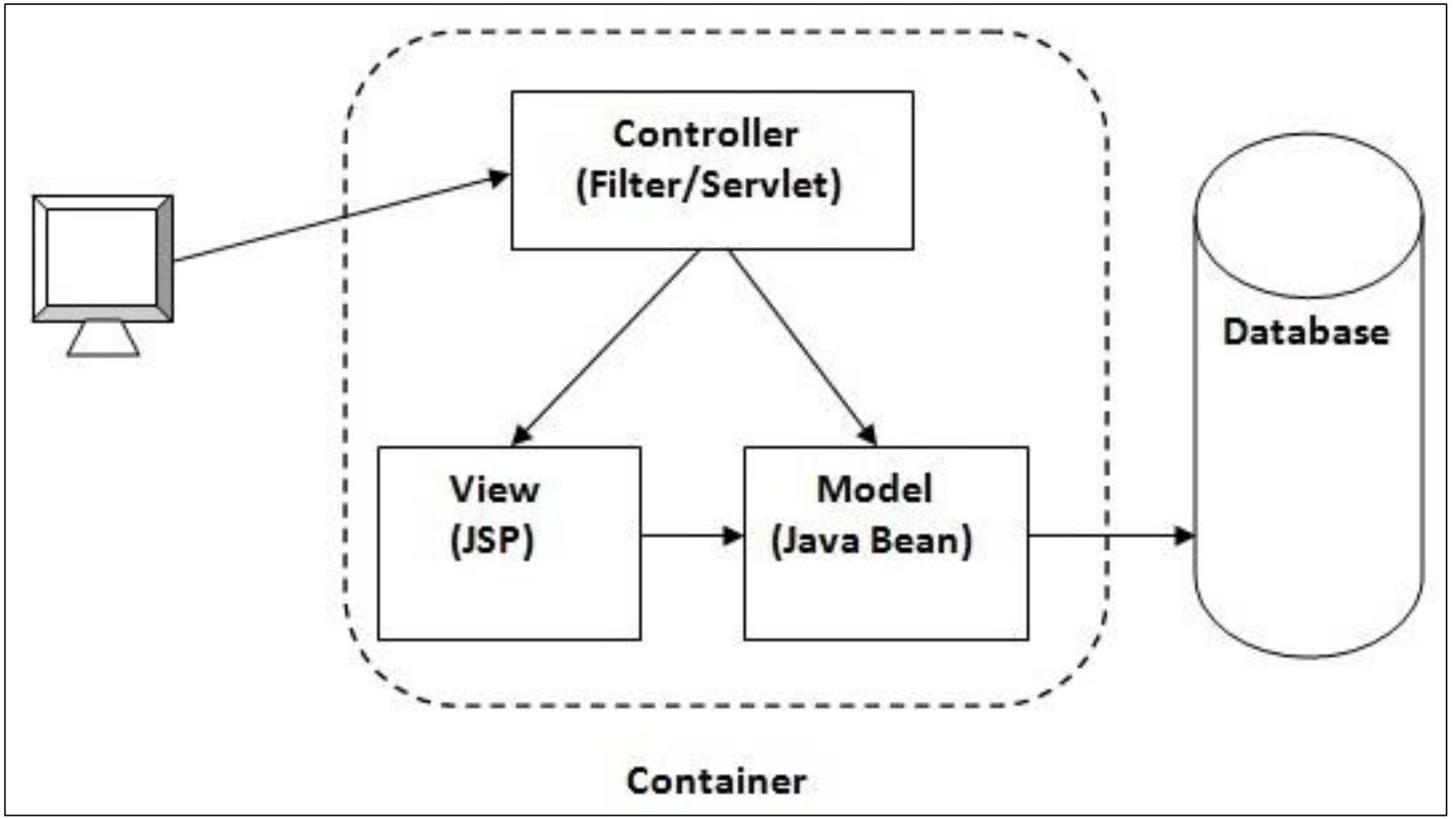
JSP Application Design with MVC

1. JSP is a technology in which the request processing, business logic and presentation logic are separated out.
2. The design of JSP is called MVC Model.
3. MVC stands for Model-View-Controller.
4. The basic idea of MVC is to separate out the 3 logics of:
    1. Modelling
    2. Viewing
    3. Controlling

MVC in JSP

- **MVC** stands for Model View and Controller. It is a **design pattern** that separates the business logic, presentation logic and data.
- **Controller** acts as an interface between View and Model. Controller intercepts all the incoming requests.
- **Model** represents the state of the application i.e. data. It can also have business logic.
- **View** represents the presentaion i.e. UI(User Interface).
- we are using servlet as a controller, jsp as a view component, Java Bean class as a model.

MVC in JSP (Cont.)



JSP Application Design with MVC

1. The **business logic** refers to the coding logic applied for manipulation of application data.
2. The **presentation logic** refers to the code written for look and feel of the web page.
3. The **request processing / controlling** is nothing but a combination of business logic and presentation logic. Request processing is done in order to generate response.

Advantages of MVC Design

1. MVC provides developer to keep **separation between business logic**, **presentation and request processing.**
2. Due to separation it is easy to change in **presentation without disturbing the business logic.**
3. **Parallel development** of code can be done.
4. The code can be **reusable**
5. It is **easy to maintain and enhance** the project using MVC architecture.

Advantages of MVC Design

6. One can attach **multiple views to model** in order **to have different presentations**.
7. The developer working on view model need not have to know about controller model and vice versa.