Unit 10
JSP
(Java Server Pages)

JSP Basics

- A typical JSP page very much looks like html page with all the html markup except that you also see Java code in it. So, in essence,

  HTML + Java = JSP
- Therefore, JSP content is a mixed content, with a mixture of HTML and Java.
- If this is the case, let me ask you a quick question. Can we save this mixed content in a file with “.html” extension?
- No we can’t, because the html formatter will also treat the Java code as plain text which is not what we want.
- We want the Java code to be executed and display the dynamic content in the page. For this to happen, we need to use a different extension which is the “.jsp” extension.
- Good. To summarize, a html file will only have html markup, and a jsp file will have both html markup and Java code. Point to be noted.

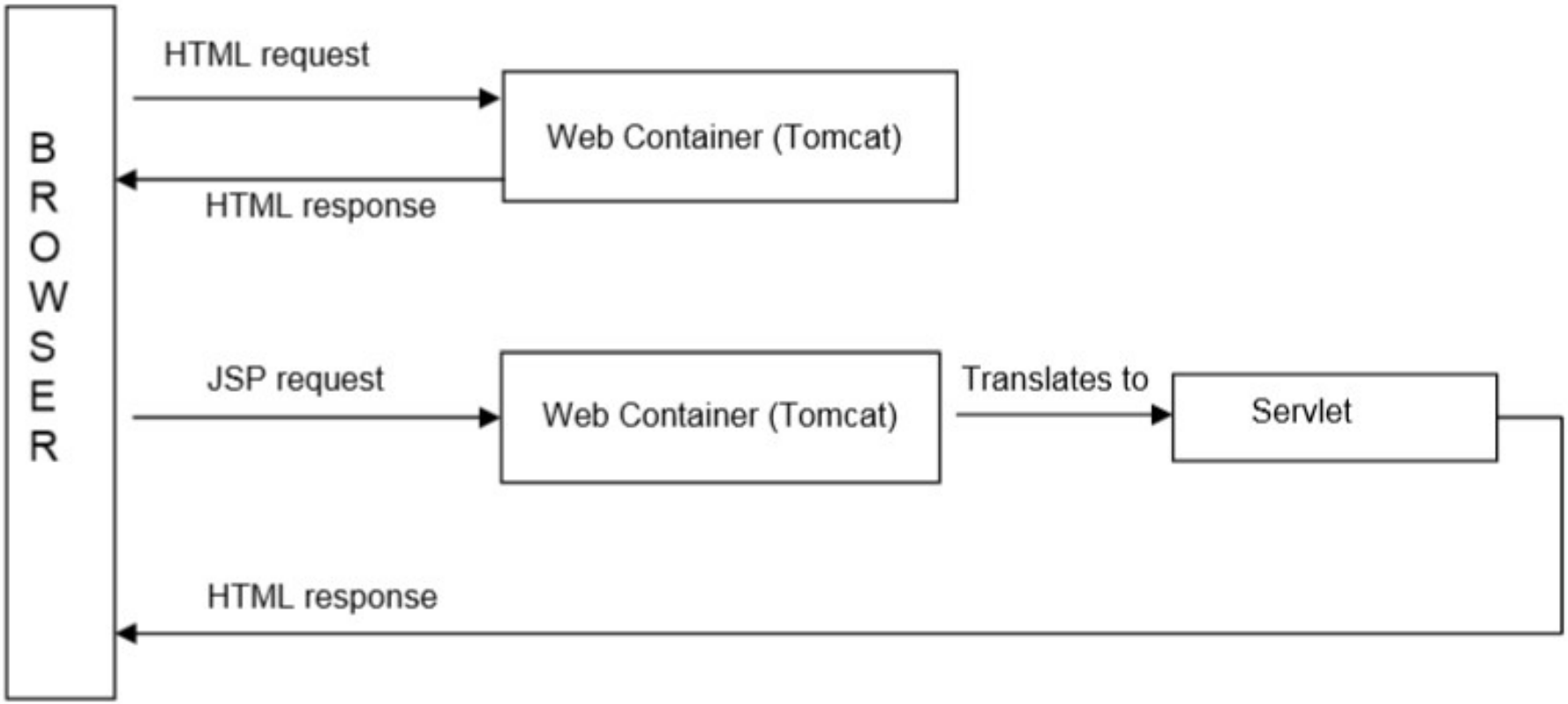


Fig: HTML and JSP page request

- The above picture should give you some idea of how a web container processes html requests and JSP requests.
- When the browser requests for html file, the web container simply responds with a html response without any processing.
- However, when the browser sends a JSP page request, the web container assumes that the JSP page might include Java code, and translates the page into a Servlet.
- The servlet then processes the Java code, and returns the complete html response including the dynamic content generated by the Java code.
- Though the web container does the translation of JSP to Servlet, we need to know how it does this translation.
- For a web container to translate JSP, it needs to identify from the JSP page, as to which is HTML markup and which is Java code.
- According to J2EE specification, a JSP page must use special symbols as placeholders for Java code.
- The web container instead of scratching its head to

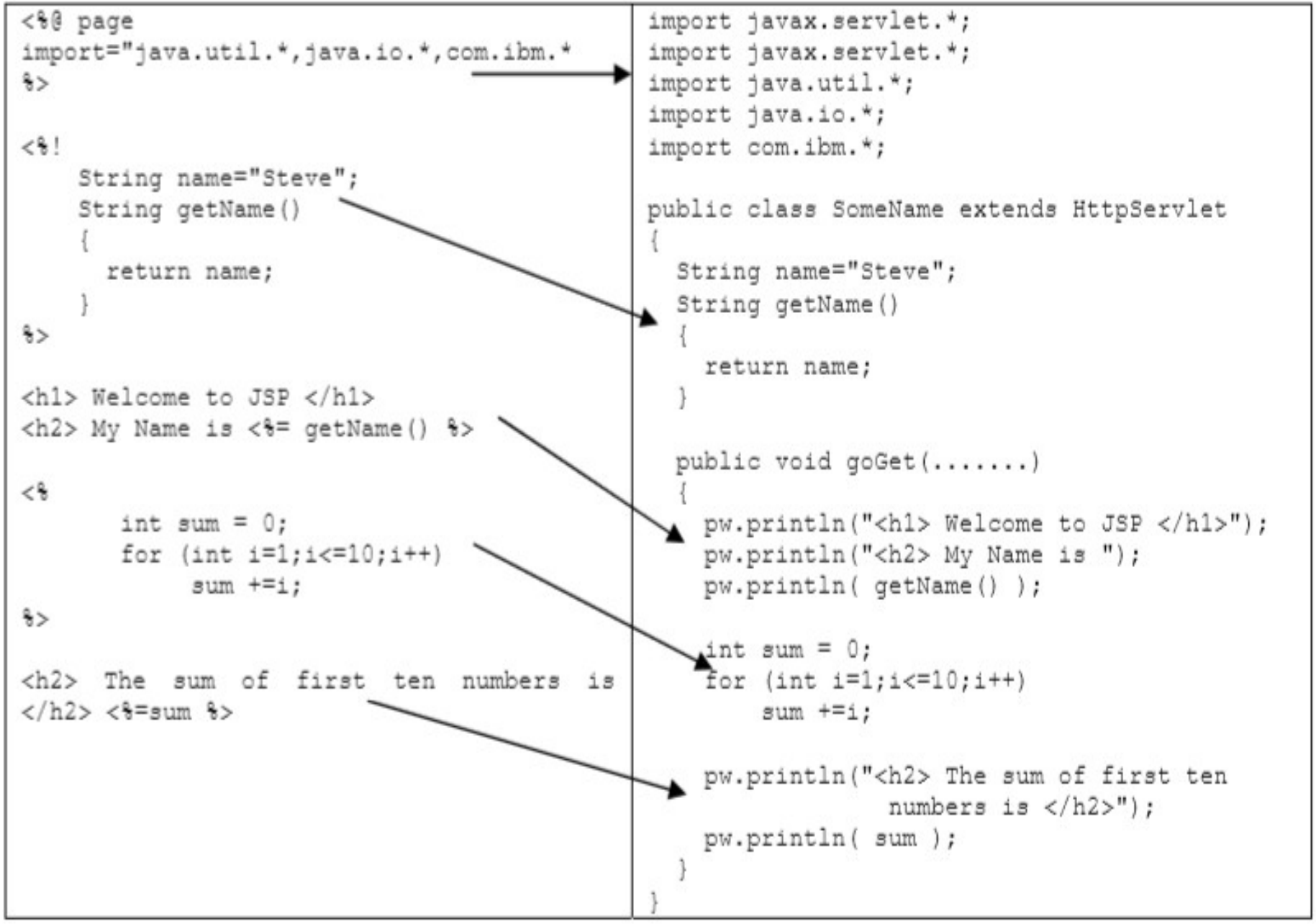
- In JSP, we use four different types of placeholders for Java code. Following table shows these placeholders along with how the web container translates the Java code with them.

JSP Code	Translated to
<%! Java Code %>	Global Variables and Methods in a Servlet

<% Java Code %>	doGet
	{
	Java Code
	}
	doPost
	{
	Java Code
	}

<%= Java Code %>	PrintWriter pw = res.getWriter(); pw.println(Java Code);
<%@ %>	Mostly as imports in Servlet

To better understand the above four place holders, let's take a sample JSP page and see how the web container translates it into a Servlet. See the following JSP and Servlet code snippets.


<%@ page
import="java.util.*,java.io.*,com.ibm.*
%>
<%!
String name="Steve";
String getName()
{
return name;
}
%>
<h1> Welcome to JSP </h1>
<h2> My Name is <%= getName() %>
<%
int sum = 0;
for (int i=1;i<=10;i++)
sum +=i;
%>
<h2> The sum of first ten numbers is
</h2> <%=sum %>
import javax.servlet.*;
import javax.servlet.*;
import java.util.*;
import java.io.*;
import com.ibm.*;
public class SomeName extends HttpServlet
{
String name="Steve";
String getName()
{
return name;
}
public void goGet(.....)
{
pw.println("<h1> Welcome to JSP </h1>");
pw.println("<h2> My Name is ");
pw.println(getName());
int sum = 0;
for (int i=1;i<=10;i++)
sum +=i;
pw.println("<h2> The sum of first ten
numbers is </h2>");
pw.println(sum);
}
}

- The above table shows exactly how the Java code in various placeholders is translated into different places in a servlet.
- Look at the rules in previous table and every thing will make sense.
- Even if you don’t understand the above translation, it’s still fine because we don’t do this translation. We are just trying to understand what the web container does.
- After all, who cares what it does. So, don’t worry. All you just have to know as a JSP developer is, how and for what purposes we use various placeholders.
- Understanding this hardly takes few minutes. So

JSP Directives

- Directives are used for declaring classes, session usage etc., and does not produce any response to the client.
- A directive uses attributes for declarations. There are 3 types of directives as listed below:
  - page directive
  - include directive
  - taglib directive

The page directive

- This directive is used to declare things that are important to the entire JSP page. The syntax for this directive is shown below
  - <%@ page attribute list %>
- Following table lists the most widely used attributes with this directive:

page Attribute	Description
`import`	Used by the current JSP for importing the resources
`session`	Used by the current JSP for session management
`errorPage`	Used to specify the error pages for the current JSP
`isErrorPage`	Used to specify the current JSP as an error page to some other JSP

- Ex 1: If the JSP page needs to import the core library classes, it uses the page directive as:
- <%@ page import=”java.util.*, java.io.*” %>
- All the resources must be separated by a comma as shown above.
- Ex 2: If the JSP page needs to use the session, then its uses page directive attribute as shown below:
- <%@ page session=”true” %>
- If the session attribute is set to true, then the page will have access to session.

- Ex 3: If a JSP page should forward to a custom error page for any exceptions within the page, the page directive will be as shown below:
- <%@ page errorPage=”Error.jsp” %>
- The web container will then forward the request to Error.jsp if it encounters exceptions within the page.
- Ex 4: If a JSP page should allow itself as an error page for other JSP pages, it should use the following page attribute:
- <%@ page isErrorPage=”true” %>
- Instead of defining one attribute per page directive, we can define all the attributes at a time as shown below:
- <%@ page import=”java.util.*” session=”true” errorPage=”Error.jsp” %>

PageDirectiveDemo.jsp

<%@ page import="java.util.*" session=”true” isErrorPage=”false”%>

<HTML>

<BODY>

<h4>Welcome to the world of JSP</h4>

This JSP uses the page directive

</BODY>

</HTML>

- Since we are writing Java code in JSP, there is 100% chance that the Java code may throw an exception.
- So, what happens when the Java code in JSP throws some exception? You'll see the very familiar page not found error.
- Such pages really freak the customers who visit the web site. Good web applications usually display custom error page rather than blank pages.
- By displaying error pages, we can let the users know what might have caused the error and give them a help desk number for customer support.
- To display custom error pages, JSP uses the page

ErrorPageDemo.jsp

```
<%@ page errorPage="Error.jsp" %>
<%
    int i=0;
    int k = 10/i;
    // This throws Arithmetic Exception
%>
```

Error.jsp

```
<%@ page isErrorPage="true" %>
<h3>
An exception is thrown by the page you're trying to access. Please don't panic and call 1-800-888-9999 for help desk
</h3>
```

Error.jsp

- Look at the ErrorPageDemo.jsp. It declares the page directive attribute errorPage to use Error.jsp as an error page incase any exception pops up.
- The Error.jsp page grants permission to use it as an error page by using the page directive attribute isErrorpage with the value set to true. This is like mutual understanding between the pages. If A needs to use B, then B should grant the permission to A. This is exactly what both the above JSP’s are doing.
- When the above URL accesses the ErrorPageDemo.jsp, it throws Arithmetic-Exception due to division by zero.
- Therefore, the web container gracefully forwards it to the linked error page Error.jsp which displays the custom error message.

The include directive

- This directive is used to include the response of another resource (JSP or html) at the point where the directive is placed in the current JSP. Its usage is shown below.

Syntax: <%@ include file=”file name” %>

```
<HTML>
<BODY>
<h4> This is the response from the current JSP page</h4>
<h3> Following is the response from another JSP </h3>

<hr/>
    <%@ include file=”/jsps/PageDirectiveDemo.jsp” %>
<hr/>
```

The taglib directive

- This directive allows the JSP page to use custom tags written in Java.
- Custom tag definitions are usually defined in a separate file called as Tag Library Descriptor.
- For the current JSP to use a custom tag, it needs to import the tag library file which is why this directive is used. Following is how taglib directive is used.
- <%@ taglib uri=”location of definition file” prefix=”prefix name” %>
- Custom tags and tag libraries are explained in detail in the later pages and we’ll see how to use this directive at that point. For now, don’t worry about it.
- Now that we know how and when to use JSP

JSP Declarations

- JSP declarations are used to declare global variables and methods that can be used in the entire JSP page. A declaration block is enclosed within <%! and %> symbols as shown below:

<%!

Variable declarations

Global methods

%>

JSPDeclarationDemo.jsp

```
<%@ page import = "java.util.Date" %>
<%!
    String getGreeting( String name){
        Date d = new Date();
        return "Hello " + name + "! It's "+ d + " and how
        are you doing today";
    }
%>
<h3> This is a JSP Declaration demo. The JSP invokes the
global method to produce the following
</h3>
<hr/>
<h3> <%= getGreeting("James Bond") %>
<hr/>
```

JSP Expressions

- Expressions in JSP are used to display the dynamic content in the page.
- An expression could be a variable or a method that returns some data or anything that returns a value back. Expressions are enclosed in <%= and %> as shown below
- <%= Java Expression %>

```
<%!
    String name="John Smith";
    String address = "1111 S St, Lincoln, NE, USA";
    String getMessage(){
        return "Your shipment has been sent to the following
        address. Thank you  for shopping at
        BuyForLess.com";
    }
%>
<h3> Your order is successfully processed. The
confirmation number is 876876
</h3>
<hr/>
<h3> <%= getMessage() %> </h3>
<h4> <%= name %>
    <%= address %>
</h4>
<hr/>
```

JSP Scriptlets

- A Scriptlet is a piece of Java code that represents processing logic to generate and display the dynamic content where ever needed in the page. Scriptlets are enclosed between <% and %> symbols. This is the most commonly used placeholder for Java code

```
<HTML>
<HEAD>
<TITLE>Tag - Methods</TITLE>
</HEAD>
<h3> This is an example using Scriplets </h3>
<%
    int sum=0;
    for(int i=1;i<=100;i++) {
        sum+=i;
    }
%>
```

```
<hr/>
The sum of first 100 numbers is <%= sum %>
<hr/>
<h3> Following is generated by the Loop
<%
for(int i=2;i<=5;i++){
     if( i % 2 == 0){
%>
      <br/><%=i %> is an even number
<%     }
else{
%>
<br/><%= i %> is an odd number
<%
 }
} // End of for loop
%>
</HTML>
```

Implicit Objects

- As the name suggests every JSP page has some implicit objects that it can use without even declaring them.
- The JSP page can readily use them for several purposes

Implicit Object	Description
request	This is an object of HttpServletRequest class. Used for reading request parameters (Widely used)
response	This is an object of HttpServletResponse class. Used for displaying the response content. This is almost never used by the JSP pages. So, don’t worry about it.
session	This is an object of HttpSession class used for session management. (Widely Used)
application	This is an object of ServletContext. Used to share data by all Web applications. Rarely used.

- Out of all the above implicit objects, only request and session objects are widely used in JSP pages.
- The request object is used for reading form data and session object is used for storing and retrieving data from session.
- Let’s see an example using request and session implicit objects.
- In this example, we will have html form post some data to a JSP page with reads the form parameters and stores them in the session.
- We will then write another JSP that reads the data from the session and displays them in the browser.

Login.jsp

```
<html>
<body>
<form action="StoreData.jsp">
<table border=1>
<tr> <td> Login Name </td>
<td> <input type="text" name="username"
size="20"/></td>
</tr>
<tr> <td> Password </td>
<td> <input type="password" name="password"
size="20"/></td>
</tr>
<tr> <td> <input type="submit"
value="Submit"/></td>
</tr>
</table>
```

StoreData.jsp

```
<%@ page session="true" %>
<%
        // Read the data from the request
        String name =
request.getParameter("username");
        String pwd =   request.getParameter("password");

        // Store the data in the session
session.setAttribute("userid",name);
session.setAttribute("password",pwd);
%>
<h3> This page read the form data and stored it in the
session.</h3>
<a href="RetrieveData.jsp">Click Here</a> to go to the
```

RetrieveData.jsp

```
<%@ page session="true" %>
<%
        // Read the data in the session
        String name =
(String)session.getAttribute( "userid");
        String pwd    =
(String)session.getAttribute("password");
%>
<h3>
The username is <%= name %> <br/>
The password is <%= pwd %>
```

- The StoreData.jsp does two things listed below:
- 1. Read the form data using the request implicit object.

```
String name = request.getParameter("username");

String pwd =   request.getParameter("password");
```

- 2. Store the form data in the session using the session implicit object.

```
session.setAttribute( "userid",name);
session.setAttribute("password",pwd);
```

- This page then provides a link to the RetriveDa
- The RetrivePage.jsp reads the data stored in the session as shown below and displays it.

```
String name =
```

Java Bean

A Java Bean is a java class that should follow following conventions:

- It should have a no-arg constructor.
- It should be Serializable.
- It should provide methods to set and get the values of the properties, known as getter and setter methods.

Why use Java Bean?

- It is a reusable software component.
- A bean encapsulates many objects into one object, so we can access this object from multiple places.
- Moreover, it provides the easy maintenance.

Employee.java

```
package beans;
public class Employee {
    private int id;
    private String name;
    public int getId() {
        return id;}
    public void setId(int id) {
        this.id = id;}
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Test.java

```
package beans;
public class Test {
    public static void main(String[] args) {
        Employee e=new Employee();
        e.setId(1);
        System.out.println(e.getId());
        e.setName("James Bond");
        System.out.println(e.getName());
    }

}
```

Java Beans in JSP

- One of the good practices while writing JSP is to isolate the presentation logic (HTML) from business logic (Java code).
- A typical JSP page should have as minimum business logic as possibly can. If this is the case, where should the business logic be?
- The business logic will be moved into external entities which will then be accessed from within JSP page. These external entities are nothing but Java beans.
- A Java bean is a simple class with getters and setters.
- Using Java beans in JSP offers whole lot of flexibility and avoids duplicate business logic.

- 1. <jsp:useBean>
- 2. <jsp:setProperty>
- 3. <jsp:getProperty>

jsp:useBean

- This action is used by the web container to instantiate a Java Bean or locate an existing bean.
- The web container then assigns the bean to an id which the JSP can use to work with it.
- The Java Bean is usually stored or retrieved in and from the specified scope.
- The syntax for this action is shown below:
- <jsp:useBean id=”bean name” class=”class name” scope=”scope name”/>

where,

- id - A unique identifier that references the instance of the bean
- class - Fully qualified name of the bean class
- scope -The attribute that defines where the bean will be stored or retrieved from; can be request or session

Scope of Bean

- **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.

- Consider the following declaration.
- <jsp:useBean id = “cus” class=”beans.Customer” scope=”session” />
- With the above declaration, following is what the web container does.
  1. Tries to locate the bean with the name cus in session scope. If it finds one, it returns the bean to the JSP.
  2. If the bean is not found, then the container instantiates a new bean, stores it in the session and returns it to the JSP.
- If the scope is session, then the bean will be available to all the requests made by the client in the same session.
- If the scope is request, then the bean will only be available for the current request only.

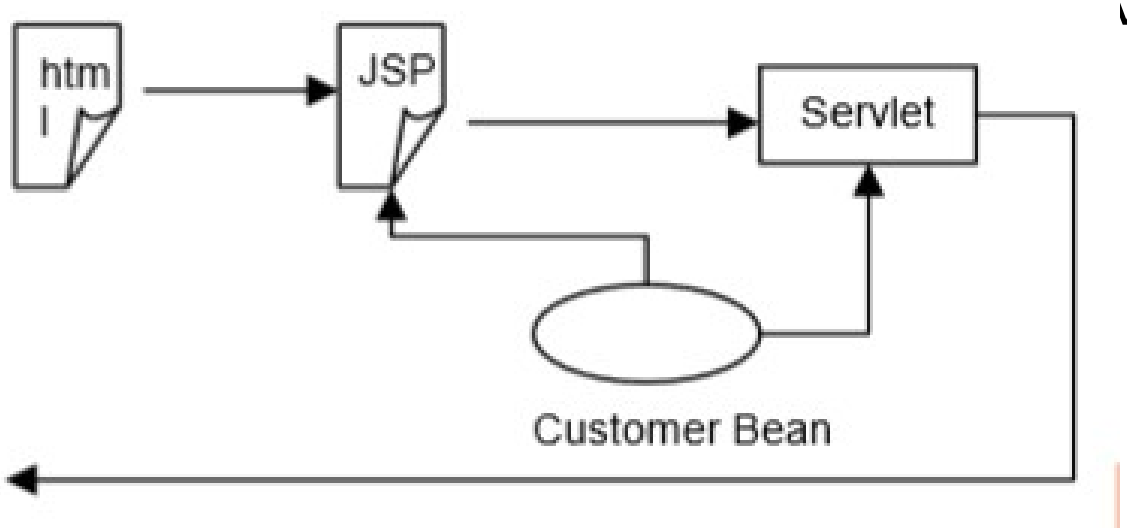
jsp:setProperty

- This action as the name suggests is used to populate the bean properties in the specified scope. Following is the syntax for this action.
- <jsp:setProperty name ="bean name" property ="property name" value= "data" />
- For instance, if we need to populate a bean whose property is firstName with a value John we use this action as shown below:
- <jsp:setProperty name "cus" property ="firstName" value= "John" />

jsp:getProperty

- This standard action is used to retrieve a specified property from a bean in a specified scope. Following is how we can use this action to retrieve the value of firstName property in a bean identified by cus in the session scope.
- <jsp:getProperty name=”cus” property=”firstName” scope=”session” />
- There are two common scenarios with using JavaBeans in JSP.
- Scenario 1: JSP collects the data from the client, populate the bean's properties and stores the bean in request or session scope. This bean will then be used by another server side component to process the data.
- Scenario 2: A Server side component loads a Java Bean with all the information and stores it in the request or session. The JSP will then retrieve the bean and displays

- In scenario 1, JSP uses the bean to collect the data into it, while in scenario 2, it uses the bean to read the data from it to display.
- The following example demonstrates scenario 1 in which a html form posts the data to a JSP page. The JSP page will then collect the data and store it in a Java Bean named Customer in session scope.
- It then p ves the bean fro it.



Calculator.java

```
package beans;
public class Calculator {
    public int cube(int m){
        return m*m*m;
    }
}
```

Calc.jsp

```
<jsp:useBean id="cal" class="beans.Calculator"/>
<%
    int m = cal.cube(5);
    out.print("cube of 5 is" + m);
%>
```

Example

Form.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <title>JSP Page</title>
    </head>
    <body>
        <form action="BeanSetter.jsp">
            Enter Full Name:<input type="text"
name="fullname" />
            <input type="submit" value="submit"/>
        </form>
    </body>
</html>
```

Test.java

```
package beans;
public class Test {
    String name;
    public String getName() {
        return name;
    }

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

BeanSetter.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <title>JSP Page</title>
    </head>
    <body>
        <jsp:useBean id="test" class="beans.Test"
scope="session" />
        <jsp:setProperty name="test" property="name"
value="<%=request.getParameter("fullname")%>" />
        <a href="BeanGetter.jsp">Click Here for
Result</a>
        <%--<%= test.getName() %>--%>
    </body>
```

BeanGetter.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <title>JSP Page</title>
    </head>
    <body>
        <jsp:useBean id="test" class="beans.Test" scope="session"/>
        Value in bean is <jsp:getProperty name="test" property="name" />
    </body>
</html>
```

DateDisplay.jsp

```
<!DOCTYPE html>
<html>
    <head>
       <title>JSP Page</title>
    </head>
    <body>
        <jsp:useBean id = "date" class = "java.util.Date"
/>
        <p>The date/time is <%= date%>
    </body>
</html>
```

CustomerForm.html

```
<html>
   <head> <title> Customer Form </title> </head>
   <body>
      <h3>Please fill in the following details and submit it
</h3> <br/>
      <form action="CustomerInfoGatherer.jsp"
method="GET">
         First Name: <input type="text" name="firstName" >
<br>
         Middle Name: <input type="text"
name="middleName">   <br>
         Last Name: <input type="text" name="lastName">
<br>
         Age: <input type="text" name="age" >   <br>
         SSN: <input type="text" name="ssn">   <br>
         City: <input type="text" name="city">  <br>
         State: <input type="text" name="state"> <br>
         Country: <input type="text" name="country"> <br>
         <input type="submit" name="Submit"/>
```

Customer.java

```
package beans;
public class Customer implements java.io.Serializable {
    String firstName;
    String middleName;
    String lastName;
    String age;
    String ssn;
    String city;
    String state;
    String country;
    public String getAge() {
        return age;}
    public void setAge(String age) {
        this.age = age;
    }
```

```
public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getCountry() {
        return country;}
    public void setCountry(String country) {
        this.country = country;
    }
    public String getFirstName() {
        return firstName;}
```

```
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
public String getLastName() {
    return lastName;
}
public void setLastName(String lastName) {
    this.lastName = lastName;
}
public String getMiddleName() {
    return middleName;
}
public void setMiddleName(String middleName) {
    this.middleName = middleName;
}
```

```
public String getSsn() {
        return ssn;
    }

    public void setSsn(String ssn) {
        this.ssn = ssn;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}
```

CustomerInfoGatherer.jsp

```
<html>
<body>
<H3>Reading the form data</H3>
<H3>Populating the bean and Storing in session </H3>

<jsp:useBean id="userInfo" class="beans.Customer"
scope="session"/>
<jsp:setProperty name="userInfo" property="firstName"
value="<%=request.getParameter("firstName")%>" />
<jsp:setProperty name="userInfo" property="middleName"
value="<%= request.getParameter("middleName")%>" />
<jsp:setProperty name="userInfo" property="lastName"
value="<%= request.getParameter("lastName")%>" />

<jsp:setProperty name="userInfo" property="age"
value="<%= request.getParameter("age")%>" />
```

```
<jsp:setProperty name="userInfo" property="ssn"
value="<%= request.getParameter("ssn")%>" />
 <jsp:setProperty name="userInfo" property="city"
value="<%= request.getParameter("city")%>" />
<jsp:setProperty name="userInfo" property="state"
value="<%= request.getParameter("state")%>" />
<jsp:setProperty name="userInfo" property="country"
value="<%= request.getParameter("country")%>" />

        <H3>Finished storing in the session </H3>
        <br/>
        <a href="CustomerInfoProcessor" >Click Here </a>
to invoke the servlet that process the bean data in
session.
    </body>
```

InfoProcessor.java

```
package myservlet;
import beans.Customer;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class InfoProcessor extends HttpServlet {
    public void doGet(HttpServletRequest req,
HttpServletResponse res) throws ServletException,
IOException {
        String responseMsg = "";
```

```
// Get the bean from session
        HttpSession session=req.getSession(true);
        Customer customer = (Customer)
session.getAttribute("userInfo");
        String fName = customer.getFirstName();
        String mName = customer.getMiddleName();
        String lName = customer.getLastName();
        String age = customer.getAge();
        String ssn = customer.getSsn();
        String city = customer.getCity();
        String state = customer.getState();
        String country = customer.getCountry();
        res.setContentType("text/html");
        PrintWriter pw = res.getWriter();
// Send the response
        pw.println(fName + "    " + mName + "    " + lName + "
" + age + "    " + ssn + "    " + city + "    " + state + "    " +
country);
```

InfoProcessor.jsp

```
<html>
   <body>
      <jsp:useBean id="userInfo" class="beans.Customer"
scope="session" />
      <jsp:getProperty name="userInfo"
property="firstName"/>
      <jsp:getProperty name="userInfo"
property="middleName"/>
      <jsp:getProperty name="userInfo"
property="lastName"/>
      <jsp:getProperty name="userInfo" property="age"/>
      <jsp:getProperty name="userInfo" property="ssn"/>
      <jsp:getProperty name="userInfo" property="city"/>
      <jsp:getProperty name="userInfo"
property="state"/>
      <jsp:getProperty name="userInfo"
```

Example 2
BadCalculator.jsp

```
<html>
   <body>
      <%
         int sum = 0;
         for (int i = 0; i <=100; i++) {
            sum += i;
         }
      %>
      <h3>The sum of first 100 natural number is<%= sum
%> </h3>
   </body>
</html>
```

GoodCalculator.jsp

```
<html>
    <body>
        <jsp:useBean id="calculator"
class="beans.SumCalculator" />
<jsp:setProperty name="calculator" property="count"
value="100" />
        The sum of first<jsp:getProperty name="calculator"
property="count"/>
        number is  <jsp:getProperty name="calculator"
property="sum"/>
    </body>
</html>
```

Beans

```
package beans;
public class SumCalculator {
    String count;
    String sum;
    public String getCount() {
        return count;
    }
    public void setCount(String count) {
        this.count = count;
    }
```

```
public String getSum() {
        int s = 0;
        int maxcount = Integer.parseInt(count);
        for (int i = 1; i <= maxcount; i++) {
            s += i;
        }
        sum = s + "";
        return sum;
    }
    public void setSum(String sum) {
        this.sum = sum;
    }
}
```

Custom Tags

- Custom Tags are introduced from JSP 1.1 specification.
- A custom tag is a user defined tag, which looks just like an XML tag in terms of representation, but conveys a special meaning to the web container when it comes across it.
- Custom tags are extremely powerful and offers a whole lot of flexibility.
- The notion of custom tag is such a big hit that you literally see them in almost every single JSP page in any real world J2EE based web application.
- To write a custom tag and use it in a JSP page, we need to follow simple process.

- A custom tag like any standard html tag will have the following things:
- 1. A tag name
- 2. One or more tag attributes
- 3. Body Content
- 4. Nested tags
- Following is how a typical custom tag looks like.

<bean:write name=”some name” property=”some prop”>

This is the body content

</bean:write>

- In the above custom tag, write is the name of the tag and name and property are the attributes of the tag.
- With custom tags it’s very important that every tag that is opened must also be closed. Failing to do so will result in a JSP translation error.

- Following is the standard process for writing a custom tag.
- 1. Write a Tag class
- 2. Define the tag class in a tag library descriptor
- 3. Import the tag library descriptor into JSP page
- 4. Finally, use the tag.

Step 1: Writing a Tag class

- A custom tag can be viewed as a “representative” of some Java code in a JSP page.
- To write a custom tag, we need to write a Java class using the standard tag extension API.
- This API has few built-in classes that we need to use to write the Tag class.

- To write a basic custom tag,
- 1. Write a class that extends TagSupport class.
- 2. Declare one instance variable per attribute of the tag, and define a getter and setter method.
- 3. Overwrite two methods namely doStartTag() and doEndTag().
- Using the above process, let’s write the following custom tag that displays the sum of numbers.
- <calc:sum count=”100”/>
- The web container should compile the above tag and produce the following html markup. This is our first goal.
- <h3>The Sum of 100 numbers is 5050</h3>

```
package customtags;
import beans.*;
import java.io.IOException;
import javax.servlet.jsp.JspException;
import static
javax.servlet.jsp.tagext.Tag.EVAL_BODY_INCLUDE;
import static javax.servlet.jsp.tagext.Tag.EVAL_PAGE;
import javax.servlet.jsp.tagext.TagSupport;
public class CalculatorTag extends TagSupport {
// This method will be called when the JSP encounters the
start of the
// tag implemented by this class.
```

```
    String count;
    public String getCount() {
        return count;
    }
    public void setCount(String count) {
        this.count = count;
    }
    public int doStartTag() throws JspException {
// This means the JSP must evaluate the contents of any
CHild tags
// in this tag;
        return EVAL_BODY_INCLUDE;
    }
// This method is called when the JSP encounters the end
of te tag
// implemented by this class
```

```
public int doEndTag() throws JspException {
// Use the bean that we already have to calculate the
sum.
        SumCalculator calc = new SumCalculator();
        calc.setCount(count);
        String sum = calc.getSum();
        try {
            pageContext.getOut().write("The Sum of first " +
count + " numbers is " + sum);
        } catch (IOException e) {
            throw new JspException("Fatal Error: HelloTag
could'nt write to the JSP out");
        }      // This return type tells the JSP page to
continue processing
// the rest of the page
        return EVAL_PAGE;
    }
}
```

- If you observe the above code, we created a class named CalculatorTag that inherits from TagSupport class, a built-in class that supports custom tags development (Rule 1).
- Since our custom tag takes count as an attribute, we defined a property (instance variable) named count and implemented both the getter and setter method (Rule 2).
- We then overwrote two methods namely doStartTag() and doEndTag() (Rule 3).
- When the web container comes across the custom tag in the JSP, it follows a systematic approach. It first calls all the setters and assigns the attribute values to the tag class properties. It then calls the doStartTag() method.

- The doStartTag() method is usually empty and always returns EVAL_BODY_INCLUDE. Where did this come from? Don’t worry. This is a constant defined in the parent class. Returning this constant means, “Hey container, please evaluate the body content if there is some, otherwise go ahead with the next step”.
- Since our tag doesn’t have any body content, the web container gracefully goes to the next step which is the invocation of doEndTag() method.
- Entering in to this method means, the web container is at the end of the tag. This is where we need to do what ever we want to have the custom tag render the desired html to JSP page.If you notice the implementation of this method, it first uses the Calculator bean class, and computes the sum as shown below:

```
Calculator calc = new Calculator();
calc.setCount(count);
String sum = calc.getSum();
```

- Once the sum is computed, it should return it back to the JSP in proper format. For the custom tag to return anything to the JSP, it need to use the pageContext object inherited from the parent class and use the write() method as shown below:

```
pageContext.getOut().write("<h3>The Sum of first
"+count+" numbers is " + sum + “</h3>”);
```

- Now that it finished sending the desired html markup to the JSP, it must return the constant EVAL_PAGE. This means, “Hey container, I am done processing the tag, so please go ahead with the rest of the page”.

Step 2: Define the tag class in the tag library descriptor.

- This is the most important step. A tag library descriptor is a standard XML file that defines all the details of the custom tag that the web container needs to process it. It defines the following information:
- 1. Name of tag
- 2. Class of the tag
- 3. Attributes of the tag
- See the following tag library descriptor for our custom tag shown below:.
- <calc:sum count=”100”/>

Mytags.tld

```
<taglib>
        <tlibversion>1.0</tlibversion>
        <jspversion>1.1</jspversion>
<shortname>examples</shortname>
        <info>Simple Library</info>
        <tag>
                <name>sum</name>

<tagclass>customtags.CalculatorTag</tagclass>
                <bodycontent>JSP</bodycontent>
                <attribute>
                        <name>count</name>
                        <required>true</required>
                        <rtexprvalue>false</rtexprvalue>

                </attribute>
        </tag>
```

- The only important element in the above XML is the one highlighted in bold.
- The elements like tlibverion, jspversion, shortname and info are used to convey some basic information to the container.
- These are the required elements. The important one is the tag element which defined the name, the tag class, and the list of attributes the tag will have.
- The above code is pretty obvious and conveys a clear picture.
- This file though contains xml content, must be saved to a file with a “.tld” extension. Save this file in the following directory as:

Step 3: Importing the TLD file into the JSP page

- Remember we talked about JSP directives at the beginning and delegated the usage if third directive named taglib to later section.
- Now is the time to use this. This directive is used to import the TLD file into the JSP page as shown below:
- <%@ taglib uri=”/WEB-INF/tlds/mytags.tld” prefix=”calc” %>
- The above directive will tell the web container something like, “Hey container, this JSP uses custom tags that are defined in mytags.tld file which has all the information to process the tags. Moreover, all the tags defined by this TLD file will be prefixed with calc” as shown below:

Step 4: Using the custom tag in the JSP page.

- <%@ taglib uri="/WEB-INF/tlds/mytags.tld" prefix="calc" %> <calc:sum count="100" />