

## Unit 7(Part 2): JSP programming

- JSP is another J2EE technology for building web applications using Java.
- JSP technology is used to create web application just like Servlet technology. It can be thought of as an extension to Servlet because it provides more functionality than servlet such as expression language, JSTL, etc.
- **Java Server Pages (JSP)** is a technology which is used to develop web pages by inserting Java code into the HTML pages by making special JSP tags.
- JSP technology is built on top of servlet technology. This is why we can call JSP as an abstraction over servlet. What does abstraction mean? In simple terms, abstraction is a simplified fine grained layer over a slightly more complex layer that makes the development faster and easier. More the abstraction, more the simplicity.
- JSP technology is used to create dynamic web applications. JSP pages are easier to maintain than a Servlet. JSP pages are opposite of **Servlets as a servlet adds HTML code inside Java code, while JSP adds Java code inside HTML using JSP tags**. Everything a Servlet can do, a JSP page can also do it.

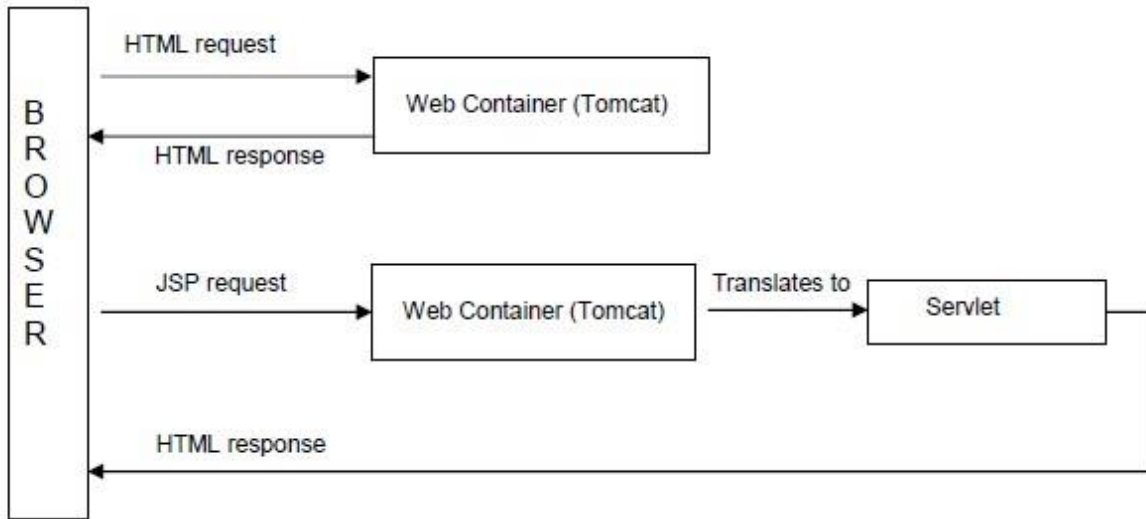
### JSP Basics

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, one question arises. Can we save this mixed content in a file with “**.html**” extension? You guessed it right. 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.

Now, look at the following figure:



*Figure 1: HTML 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.
- ✓ 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 identify Java code, simply uses the special symbols to identify it. This is a contract between JSP and the Web container.
- ✓ Let's understand what these special symbols are. 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.

<pre> &lt;%@ page import="java.util.*,java.io.*,com.ibm.* %&gt;  &lt;%!     String name="Steve";     String getName()     {         return name;     } %&gt;  &lt;h1&gt; Welcome to JSP &lt;/h1&gt; &lt;h2&gt; My Name is &lt;%= getName() %&gt;  &lt;%     int sum = 0;     for (int i=1;i&lt;=10;i++)         sum +=i; %&gt;  &lt;h2&gt; The sum of first ten numbers is &lt;/h2&gt; &lt;%=sum %&gt; </pre>	<pre> 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 doGet(.....)     {         pw.println("&lt;h1&gt; Welcome to JSP &lt;/h1&gt;");         pw.println("&lt;h2&gt; My Name is ");         pw.println( getName() );          int sum = 0;         for (int i=1;i&lt;=10;i++)             sum +=i;          pw.println("&lt;h2&gt; The sum of first ten                     numbers is &lt;/h2&gt;");         pw.println( sum );     } } </pre>
---	--

☞ **JSP : Java Server Pages**

☞ **JSP is a technology for building web applications using Java**

☞ **JSP is tag based**

☞ **JSP = HTML + Java Code**

☞ **Translation process and compilation process is automatically done by web container**

## Lifecycle of JSP

A JSP page is converted into Servlet in order to service requests. The translation of a JSP page to a Servlet is called Lifecycle of JSP. JSP Lifecycle is exactly same as the Servlet Lifecycle, with one additional first step, which is, translation of JSP code to Servlet code. Following are the JSP Lifecycle steps:

1. Translation of JSP to Servlet code.
2. Compilation of Servlet to bytecode.
3. Loading Servlet class.
4. Creating servlet instance.
5. Initialization by calling *jspInit()* method
6. Request Processing by calling *\_jspService()* method
7. Destroying by calling *jspDestroy()* method

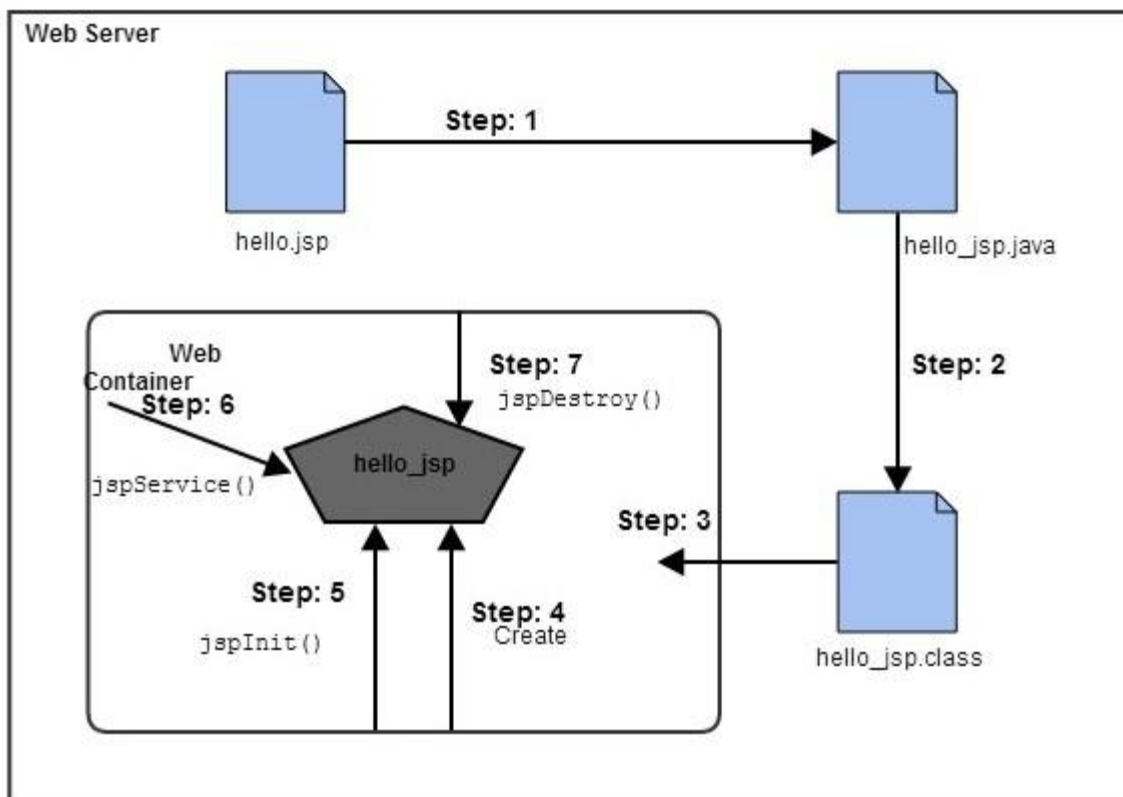


Figure 2: JSP Life Cycle

**Translation** – JSP pages doesn't look like normal java classes, actually JSP container parse the JSP pages and translate them to generate corresponding servlet source code. If JSP file name is `home.jsp`, usually its named as `home_jsp.java`.

**Compilation** – If the translation is successful, then container compiles the generated servlet source file to generate class file.

**Class Loading** – Once JSP is compiled as servlet class, its lifecycle is similar to servlet and it gets loaded into memory.

**Instance Creation** – After JSP class is loaded into memory, its object is instantiated by the container.

**Initialization** – The JSP class is then initialized and it transforms from a normal class to servlet. After initialization, ServletConfig and ServletContext objects become accessible to JSP class.

**Request Processing** – For every client request, a new thread is spawned with ServletRequest and ServletResponse to process and generate the HTML response.

**Destroy** – Last phase of JSP life cycle where it's unloaded into memory.

Web Container translates JSP code into a servlet class source (.java) file, then compiles that into a java servlet class. In the third step, the servlet class bytecode is loaded using classloader. The Container then creates an instance of that servlet class.

The initialized servlet can now service request. For each request the Web Container call the `_jspService()` method. When the Container removes the servlet instance from service, it calls the `jspDestroy()` method to perform any required clean up.

### What happens to a JSP when it is translated into Servlet

Let's see what really happens to JSP code when it is translated into Servlet. The code written inside `<% %>` is JSP code.

```
<html>

<head><title>My First JSP Page</title> </head>

    <%          int
count = 0;
    %>

    <body>

        Page Count is:

        <% out.println(++count); %>

    </body>
</html>
```

The above JSP page(hello.jsp) becomes this Servlet

```
public class hello_jsp extends HttpServlet
{
    public void _jspService(HttpServletRequest request,
        HttpServletResponse response) throws
        IOException, ServletException
    {
        PrintWriter out =
        response.getWriter();
        response.setContentType("text/html");
        out.write("<html><body>");          int
        count=0;          out.write("Page count is:");
        out.print(++count);
        out.write("</body></html>");

    }

}
```

*This is just to explain, what happens internally. As a JSP developer, you do not have to worry about how a JSP page is converted to a Servlet, as it is done automatically by the web container*

**JSP Scripting Elements**

☞ JSP Scripting elements are written inside `<% %>` tags. These code inside `<% %>` tags are processed by the JSP engine during translation of the JSP page. Any other text in the JSP page is considered as HTML code or plain text.

Scripting Element	Syntax
Comment	<code>&lt;%-- comment --%&gt;</code>
Directive	<code>&lt;%@ directive %&gt;</code>
Declaration	<code>&lt;%! declarations %&gt;</code>
Scriptlet	<code>&lt;% scriptlets %&gt;</code>
Expression	<code>&lt;%= expression %</code>



## 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.
- ✎ Directives can have a list of attributes defined in key-value pairs and separated by commas.
- ✎ There are 3 types of directives as listed below:

✓ **page directive** ✓ **include directive** ✓ **taglib directive**

Directive	Description
<code>&lt;%@ page ... %&gt;</code>	Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
<code>&lt;%@ include ... %&gt;</code>	Includes another file.
<code>&lt;%@ taglib ... %&gt;</code>	Declares a tag library containing custom actions which can be used in the page.

### 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
<code>import</code>	Used by the current JSP for importing the resources
<code>session</code>	Used by the current JSP for session management
<code>errorPage</code>	Used to specify the error pages for the current JSP
<code>isErrorPage</code>	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" %>
```

#### **Example:**

**//File:PageDirectiveDemo.jsp**

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

```
<HTML>
```

```
<HEAD><TITLE>PageDirectiveDemo</TITLE></HEAD>
```

```
<BODY>
```

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

```
This JSP uses the page directive
```

```
</BODY>
```

```
</HTML>
```



### 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.

`<%@ include file="file_name" %>` **Example:**

**File: IncludeDirectiveDemo.jsp**

```
<HTML>

  <HEAD><TITLE>IncludeDirectiveDemo</TITLE></HEAD>

  <BODY>

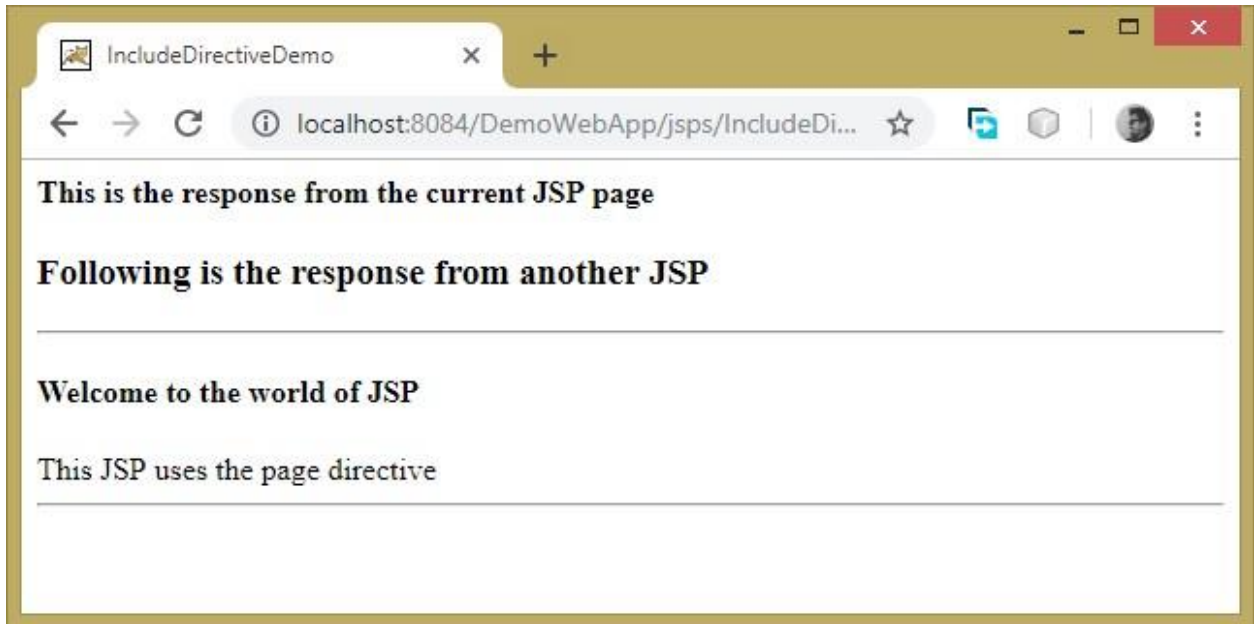
    <h4> This is the response from the current JSP page</h4>
    <h3> Following is the response from another JSP </h3>
    <hr/>

    <%@ include file='/jsp/PageDirectiveDemo.jsp' %>

    <hr/>

  </BODY>

</HTML>
```



### 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"
%>
```

### 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
```

```
%>
```

Example:

```
<html>

  <head><title>JSPDeclarationDemo</title></head>

  <body>

<%@ page import = "java.util.Date" %> <%!
String getGreeting( String name){ Date
d = new Date();
return "Namaste " + 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>

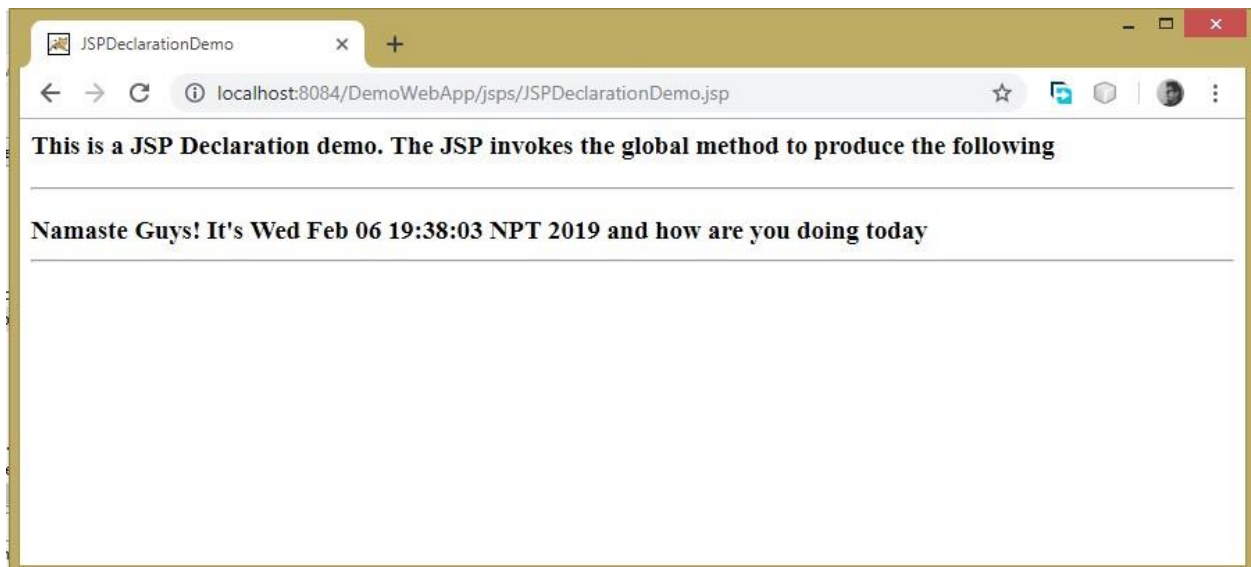
<hr/>

<h3> <%= getGreeting("Guys") %>

<hr/>

</body>

</html>
```



## 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 %>`**

### Example

```
<html>

  <head><title> JSP Expression Demo </title> </head>

  <body>

    <%!

      String str="We are students of TU";

      int fact(int n){          int result;
      if(n==0){                return 1;
      }                        else{          result
      = n*fact(n-1);

                                return result;
                                }
      }

    %>

    <h2>This is to demonstrate JSP Expressions</h2>

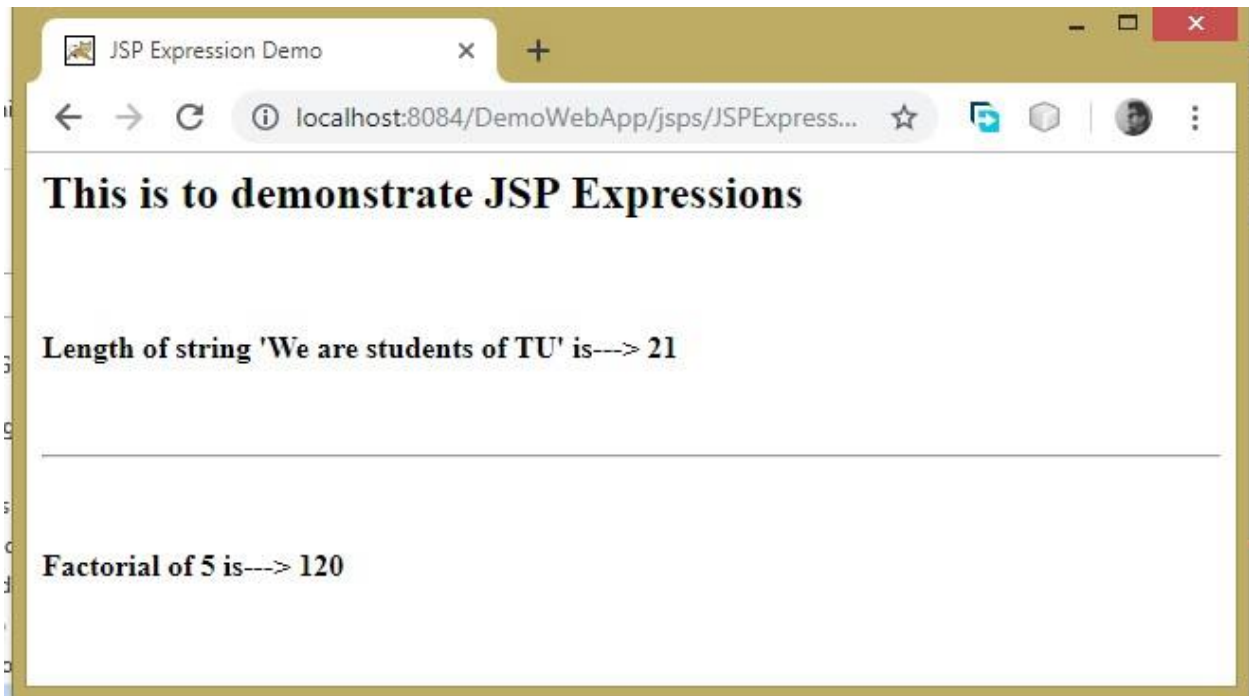
    <br/>

    <h4>Length of thestring '<%=str%>' is--->
    <%=str.length() %>
    </h4>

    <br/><hr/><br/>

    <h4> Factorial of 5 is---> <%=fact(5) %></h4>

  </body>
</html>
```



## 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.

This JSP Scripting Element allows you to put in a lot of Java code in your HTML code. This Java code is processed top to bottom when the page is the processed by the web server. Here the result of the code isn't directly combined with the HTML rather you have to use "out.println()" to show what you want to mix with HTML.

```
<html>

  <head><title>JSP Scriptlet Demo</title></head>

  <body>

    <h2> This is an example using JSP Scriptlets</h2>
<br/><hr/><h3>The multiplication table of 7 </h3>

    <%  for(int i =1; i<=10;i++){
out.print("7 * "+ i+ "= "+7*i+"<br/>");
        } %>

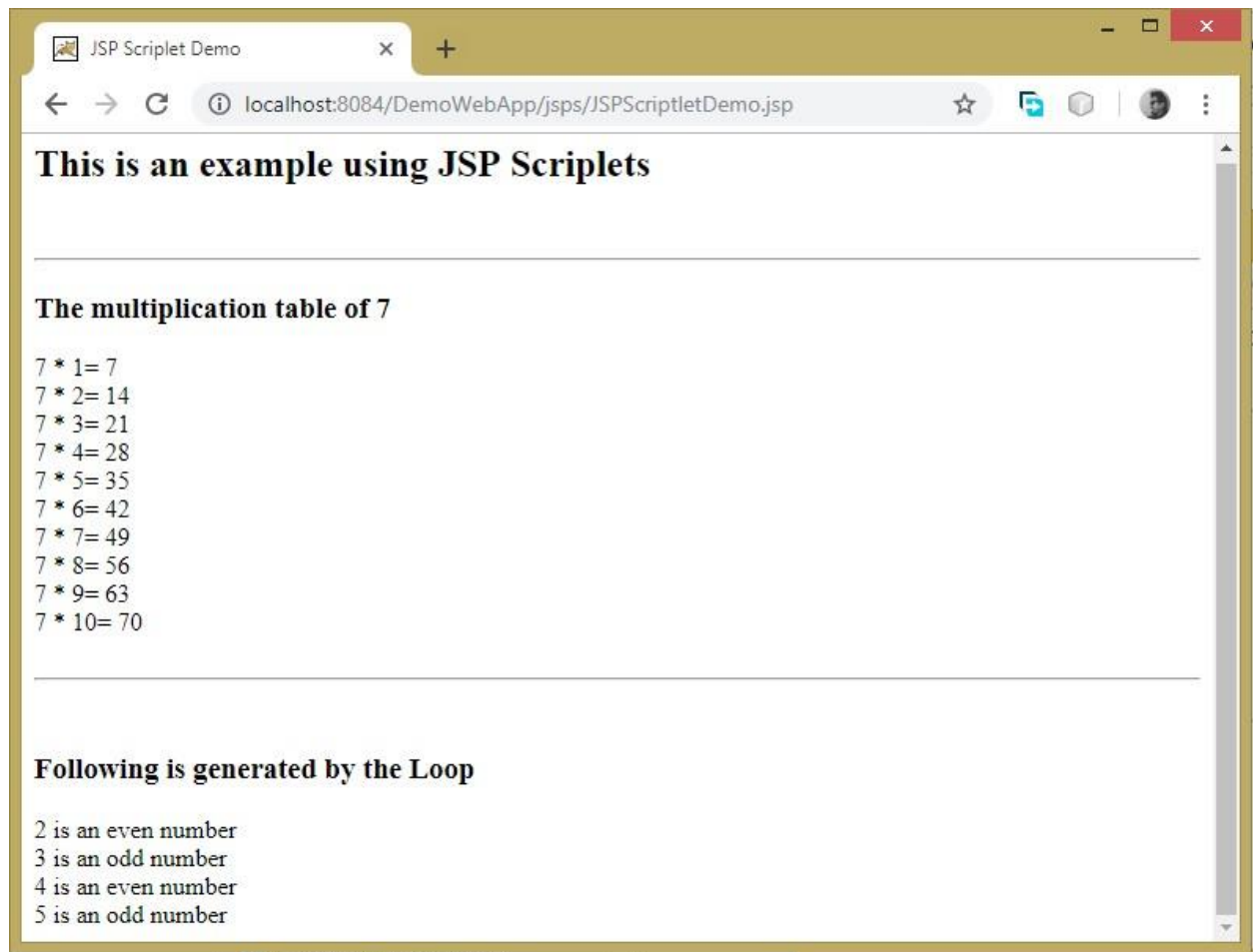
    <br/><hr/><br/>

    <h3> Following is generated by the Loop</h3>
```

```

        <% for(int i=2;i<=5;i++){           if( i % 2
== 0){           out.print(i + "  is an even number
<br/>");
        }           else{           out.print(i + "
is an odd number <br/>");
        }
    } %>
</body>
</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.



JSP implicit objects are created during the translation phase of JSP to the servlet. These objects can be directly used in scriptlets that goes in the service method. They are created by the container automatically, and they can be accessed using objects.

S.No.	Object & Description
1	<b>request</b> This is the <b>HttpServletRequest</b> object associated with the request.
2	<b>response</b> This is the <b>HttpServletResponse</b> object associated with the response to the client.
3	<b>out</b> This is the <b>PrintWriter</b> object used to send output to the client.
4	<b>session</b> This is the <b>HttpSession</b> object associated with the request.
5	<b>application</b> This is the <b>ServletContext</b> object associated with the application context.
6	<b>config</b> This is the <b>ServletConfig</b> object associated with the page.
7	<b>pageContext</b> This encapsulates use of server-specific features like higher performance <b>JspWriters</b> .
8	<b>page</b> This is simply a synonym for <b>this</b> , and is used to call the methods defined by the translated servlet class.

9	<p><b>Exception</b></p> <p>The <b>Exception</b> object allows the exception data to be accessed by designated JSP.</p>
---	--

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.

### JSP out implicit object

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(); But
```

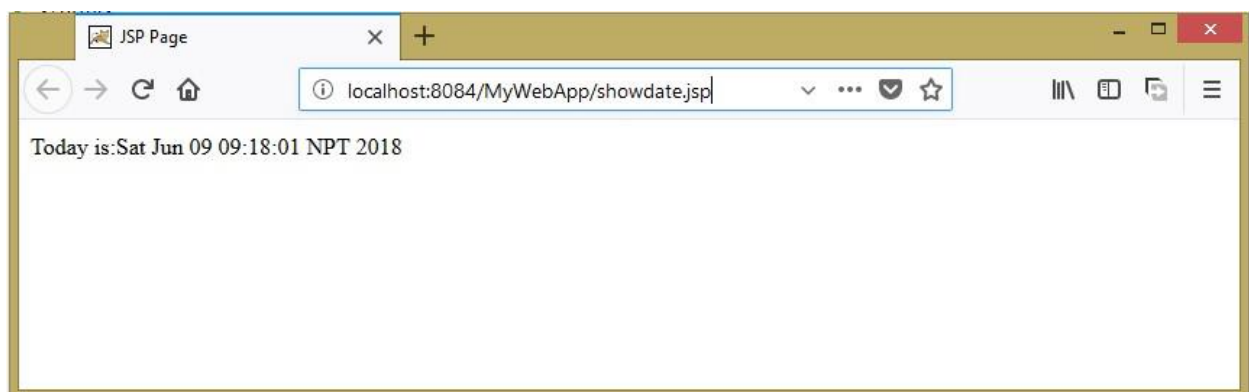
in JSP, you don't need to write this code.

### Example:

In this example we are simply displaying date and time

### showdate.jsp

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



## JSP request implicit object

The 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.

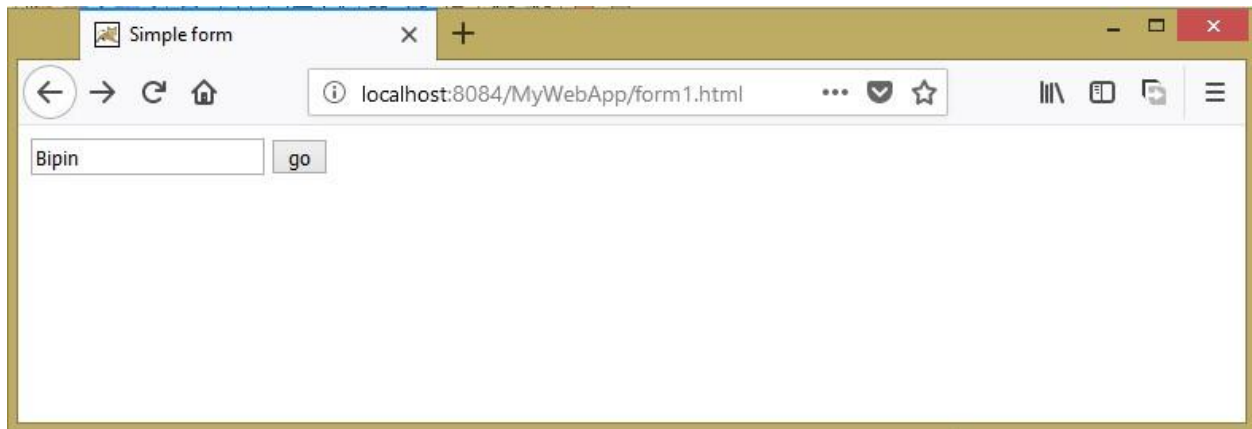
It can also be used to set, get and remove attributes from the jsp request scope.

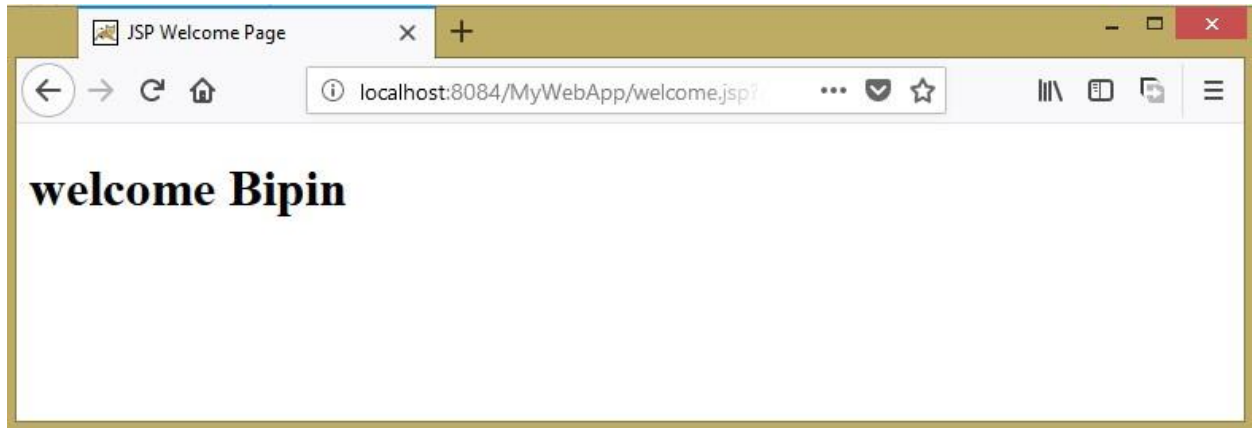
Let's see the simple example of request implicit object where we are printing the name of the user with welcome message. **form1.html**

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

## welcome.jsp

```
<h1>    <%    String name=request.getParameter("uname");
out.print("welcome "+name);    %>    </h1>
```





### JSP response implicit object

In 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.

Let's see the example of response implicit object where we are redirecting the response to the 'wordlover'. **form2.html**

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

### redirect.jsp

```
<body>

    <%
response.sendRedirect("http://www.wordloverbipin.wordpress.com");
%>

</body>
```

**JSP config implicit object**

In 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.

**JSP application implicit object**

In 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 configuration 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.

**session implicit object**

In JSP, session is an implicit object of type HttpSession. The Java developer can use this object to set, get or remove attribute or to get session information.

**pageContext implicit object**

In 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.

**exception implicit object**

In 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.

**page implicit object**

In 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

```
<% (HttpServletRequest)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 Object Scope**

The availability of a JSP object for use from a particular place of the application is defined as the scope of that JSP object. Every object created in a JSP page will have a scope. Object scope in JSP is segregated into four parts and they are page, request, session and application.

#### **Page Scope**

‘page’ scope means, the JSP object can be accessed only from within the same page where it was created. The default scope for JSP objects created using <jsp:useBean> tag is page. JSP implicit objects out, exception, response, pageContext, config and page have ‘page’ scope.

#### **Request Scope**

A JSP object created using the ‘request’ scope can be accessed from any pages that serves that request. More than one page can serve a single request. The JSP object will be bound to the request object. Implicit object request has the ‘request’ scope.

#### **Session Scope**

‘session’ scope means, the JSP object is accessible from pages that belong to the same session from where it was created. The JSP object that is created using the session scope is bound to the session object. Implicit object session has the ‘session’ scope.

#### **Application Scope**

A JSP object created using the ‘application’ scope can be accessed from any pages across the application. The JSP object is bound to the application object. Implicit object application has the ‘application’ scope.

### **Java Beans**

JavaBeans are introduced in 1996 by Sun Microsystems and defined as:

**“A JavaBean is reusable, platform independent component that can be manipulated visually in a builder tool.”**

In computing, based on the Java Platform, JavaBeans are classes that encapsulate many objects into a single object (the bean). Builder tool enables you to create and use beans for application

development purpose. In simple words JavaBean is nothing but a Java class. When these JavaBeans are used in other applications, the internal working of such components are hidden from the application developer

- A Java Bean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.
- A Java Bean is a java class that should follow following conventions:  
(Unique characteristics that distinguish a Java Bean from other Java classes)
  - 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.
- According to Java white paper, 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.
- JavaBeans makes it easy to reuse software components. Developers can use software components written by others without having to understand their inner workings. To understand why software components are useful, think of a worker assembling a car. Instead of building a radio from scratch, for example, she simply obtains a radio and hooks it up with the rest of the car.
- Beans are important because JavaBeans helps in building a complex system from the parts of software components. So, when we talk of JavaBeans, we are basically talking about the architecture that adheres to the software component model standard and how JavaBeans are integrated and incorporated to become a part of the whole subsystem.

Keep in mind:

- ✗ Java bean class name ends with Bean at last (eg: NewBean, MyBean, TestBean, CircleBean ,etc) (Its not hard and fast rule but general convention)
- ✗ Bean Class should be public
- ✗ Bean Class should implement java.io.Serializable Interface (good practice)
- ✗ Every variable of bean class are private
- ✗ Should Contain getter and setter methods and they are public
- ✗ Should contain no argument constructor ( default constructor is also no arg constructor)

### JavaBeans Properties

- A **JavaBean property** is a named attribute that can be accessed by the user of the object.

The attribute can be of any Java data type, including the classes that you define.

- A JavaBean property may be **read**, **write**, **read only**, or **write only**. JavaBean properties are accessed through two methods in the JavaBean's implementation class –

S.No.	Method & Description
1	<b>getPropertyName()</b> For example, if property name is <b>firstName</b> , your method name would be <b>getFirstName()</b> to read that property. This method is called <b>accessor</b> .
2	<b>setPropertyName()</b> For example, if property name is <b>firstName</b> , your method name would be <b>setFirstName()</b> to write that property. This method is called <b>mutator</b> .

- A **read-only** attribute will have only a **getPropertyName()** method, and a **write-only** attribute will have only a **setPropertyName()** method.

### Following are some advantages of JavaBeans:

- Reusability in different environments.
- JavaBeans are dynamic, can be customized.
- Can be deployed in network systems

### JavaBeans Example

Consider a **StudentBean** class with few properties

```
package beansexample; public class StudentBean implements
java.io.Serializable {    private String name = null;
private String address = null;    private int age = 0;
public StudentBean() {
    }    public String
getName() {        return
name;
    }    public String
getAddress() {        return
address;
    }    public int
getAge() {        return
age;
```



```

        }    public void setName(String
name){        this.name = name;
        }    public void setAddress(String
address){        this.address = address;
        }    public void setAge(int
age){        this.age = age;
        }
    }
}

```

### How to access the java bean class?

□ To access the java bean class, we should use getter and setter methods.

```

package beansexample; public class Test{
public static void main(String args[]){
    StudentBean s=new StudentBean();//object is created
//setting value to the object
    s.setName("Sushma");
    s.setAddress("Lalitpur");
    s.setAge(19);
    //getting values from the object
String name= s.getName();        String
address= s.getAddress();        int age
= s.getAge();        //printing the
values

    System.out.println(name+ " is from " + address +" and she is
" + age+ " years old.");
    }
}

```

## Using Java beans in JSP

- Using Java beans in JSP offers whole lot of flexibility and avoids duplicate business logic.  
JSP technology uses standard actions for working with beans
- Following are the three standard actions for working with Java beans:
  - **<jsp:useBean>** ▪ **<jsp:setProperty>** ▪ **<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 (widely used) 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. The default scope for JSP objects created using **<jsp:useBean>** tag is page

### **Example**

```
<html>

<head>

<title>useBean Example</title>
```

```

</head>

<body>

    <jsp:useBean id = "date" class = "java.util.Date" />

    <h5>The date/time is <%= date %></h5>

</body>

</html>

```

### **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" /> Example:
```

### **//File CustomerBean.java**

```

package somebeans; import java.io.Serializable;

public class CustomerBean implements Serializable {

    private String name;      private String address;

        private String phone;

    private String age;      public

    CustomerBean() {

        }      public String

    getName() {      return name;

    }      public String

    getAddress() {      return

    address;

```

```

    }    public String getPhone(){
return phone;    }    public String
getAge(){        return age;    }
public void setName(String name){
this.name = name;

    }    public void setAge(String
age){        this.age = age;

    }    public void setAddress(String
address){        this.address = address;

    }    public void setPhone(String
phone){        this.phone = phone;

    }

```

**//File: gatherCustomerInfo.jsp**

```

<html>

    <head>

        <title>get and set properties Example</title>

    </head>

    <body>

        <jsp:useBean id = "customer" class =
"somebeans.CustomerBean">        </jsp:useBean>

        <jsp:setProperty name = "customer" property = "name"
value = "Nikita"/>

        <jsp:setProperty name = "customer" property = "age"
value = "18"/>

        <jsp:setProperty name = "customer" property = "address"
value = "Patan"/>

```

```
<jsp:setProperty name = "customer" property = "phone"
value = "01-5678900"/>

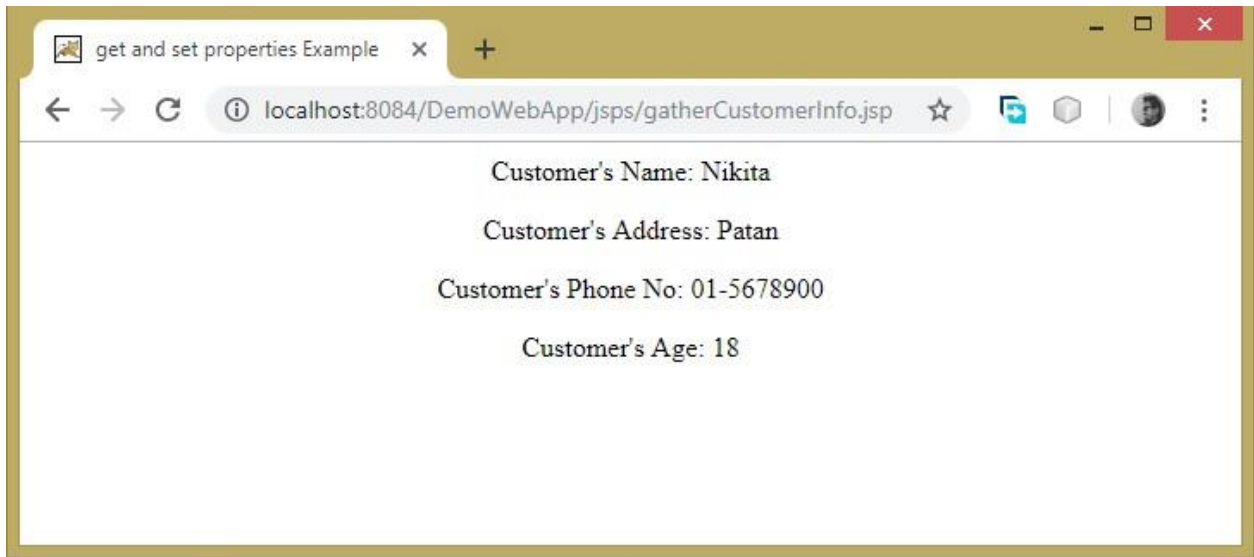
<div align='center'>
    <p>Customer's Name:
        <jsp:getProperty name = "customer" property = "name"/>
    </p>

    <p>Customer's Address:
        <jsp:getProperty name = "customer" property =
"address"/>
    </p>

    <p>Customer's Phone No:
        <jsp:getProperty name = "customer" property = "phone"/>
    </p>

    <p>Customer's Age:
        <jsp:getProperty name = "customer" property = "age"/>
    </p>
</div>
</body>
</html>
```

**Output**



**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 the information to the client.

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.

**Example.**

**Following files are included**

1. **studentForm.jsp** (form to take values from browser)
2. **StudentBean.java** ( Java Bean )
3. **loadStudentData.jsp** (uses jsp:useBean, jsp:setProperty and stores the bean in session scope and forwards the request to **displayStudentInfo.jsp** page)
4. **displayStudentInfo.jsp** (uses jsp:useBean, jsp:getProperty and extracts the bean data and displays information to client. )

**//File: studentForm.jsp**

```
<html>
<head>
    <title>StudentForm</title>
</head>
```

```

<body>
  <div align ='center'>
    <h3> Please provide the following detail </h3>
    <form action="loadStudentData.jsp" method="POST">
      <table>
        <tr>
          <td>Roll No.: </td>
          <td><input type="text" name="student_roll"/>
        </tr>
        <tr>
          <td>Name : </td>
          <td><input type="text" name="student_name"/>
        </tr>
        <tr>
          <td></td>
          <td><input type="submit" value="Submit"/>
        </tr>
      </table>
    </form>
  </div>
</body>
</html>

```

**//File: StudenBean.jsp**

```

package somebeans; import java.io.Serializable;
public class StudentBean implements Serializable {
private int rollNo;      private String name;

```

```

        public
StudentBean() {
    }
    public void
setRollNo(int rollNo){
this.rollNo=rollNo;
    }
    public void
setName(String name) {
this.name=name;
    }
    public int
getRollNo() {
    return
rollNo;
    }

    public String getName(){
return name;
    }
}

```

**//File: loadStudentData.jsp**

```

<html>

    <body>

        <jsp:useBean id = "std" class = "somebeans.StudentBean"
scope="session" ></jsp:useBean>

            <jsp:setProperty name = "std" property = "rollNo"
value=
'<%=Integer.parseInt(request.getParameter("student_roll"))%>' />

            <jsp:setProperty name = "std" property = "name" value
= '<%= request.getParameter("student_name")%>' />

        <%

```



```

        RequestDispatcher rd =
request.getRequestDispatcher("displayStudentData.jsp");
rd.forward(request, response);

    %>
</body>

```

**//File: displayStudentInfo.jsp**

```

<html>

    <head>

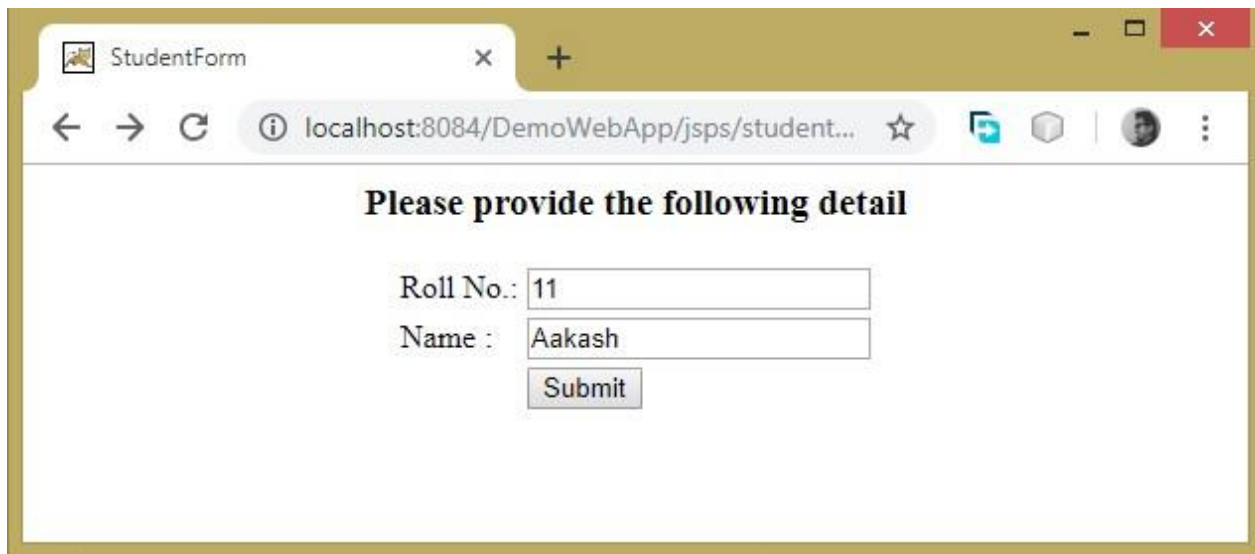
        <title>StudentInfo</title>
    </head>
    <body>
        <h1>Student's  Information </h1>
        <jsp:useBean id="std" class="somebeans.StudentBean"
scope="session" ></jsp:useBean>
        <p>
            <b>Roll No. : </b>
            <jsp:getProperty name="std" property="rollNo" />
        </p>
        <p>
            <b>Name of Student : </b>

```

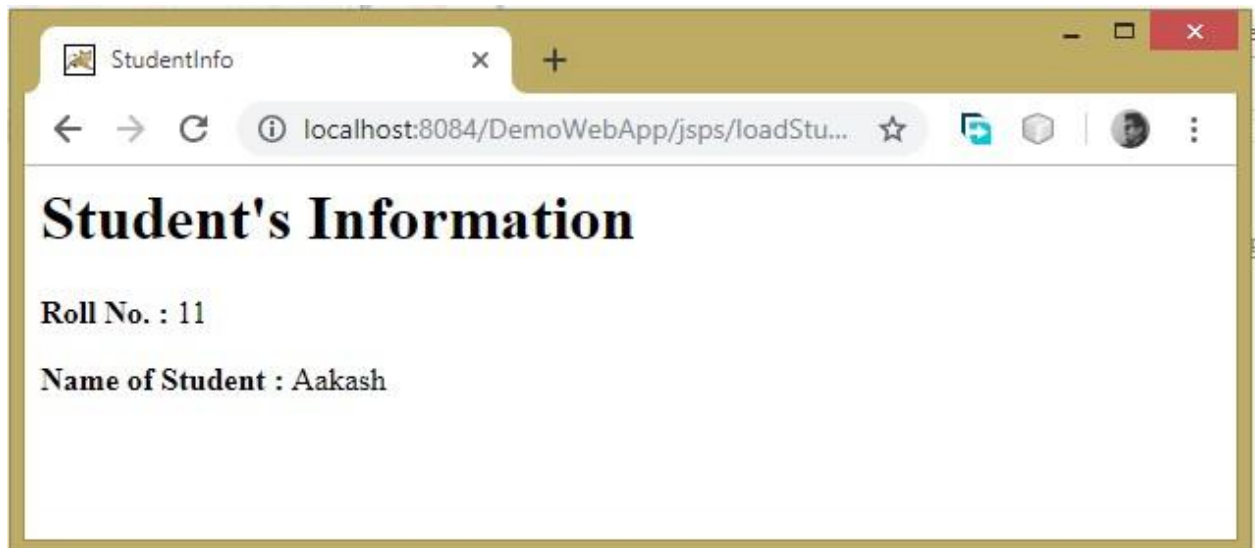
```
<jsp:getProperty name = "std" property = "name" />  
</p>
```

```
</body>
```

```
</html>
```



A screenshot of a web browser window titled "StudentForm". The address bar shows "localhost:8084/DemoWebApp/jsps/student...". The page content displays the text "Please provide the following detail" in bold. Below this, there are two input fields: "Roll No.:" with the value "11" and "Name :" with the value "Aakash". A "Submit" button is located below the name field.



A screenshot of a web browser window titled "StudentInfo". The address bar shows "localhost:8084/DemoWebApp/jsps/loadStu...". The page content displays the text "Student's Information" in large bold font. Below this, it shows "Roll No. : 11" and "Name of Student : Aakash" in bold text.