This will cover Servlets and JSP+ chapters of "Advanced Java Programming" –Unit: 7 (BSc. CSIT, TU)

Introduction to Servlets

Java Servlets are programs that run on a Web or Application server and act as a middle layer between a requests coming from a Web browser or other HTTP client and databases or applications on the HTTP server.

Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

A servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed via a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by Web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes. The *javax.servlet* and *javax.servlet.http* packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods. When implementing a generic service, you can use or extend the **GenericServlet** class provided with the Java Servlet API The **HttpServlet** class which provides methods, such as *doGet* and *doPost,* for handling HTTP-specific services

Servlet technology is used to create web application (resides at server side and generates dynamic web page).

Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was popular as a server-side programming language. But there was many disadvantages of this technology.

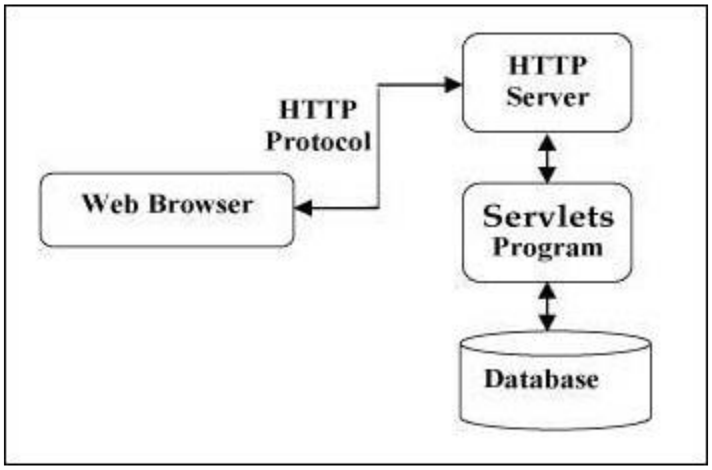


Figure 1: Position of servlets

There are many interfaces and classes in the servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse etc.

Servlet can be described in many ways, depending on the context.

- ✓ Servlet is a technology i.e. used to create web application.
- ✓ Servlet is an API that provides many interfaces and classes including documentations.
- ✓ Servlet is an interface that must be implemented for creating any servlet.
- ✓ Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any type of requests.
- ✓ Servlet is a web component that is deployed on the server to create dynamic web page

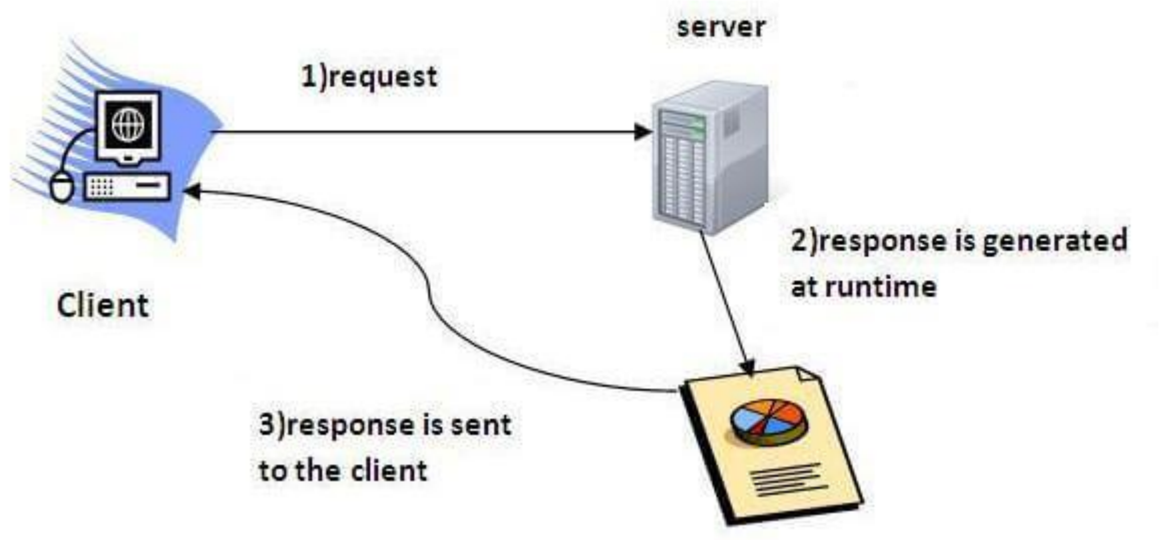

server
1)request
Client
2)response is generated at runtime
3)response is sent to the client

Figure 2: Client Server Architecture in Web

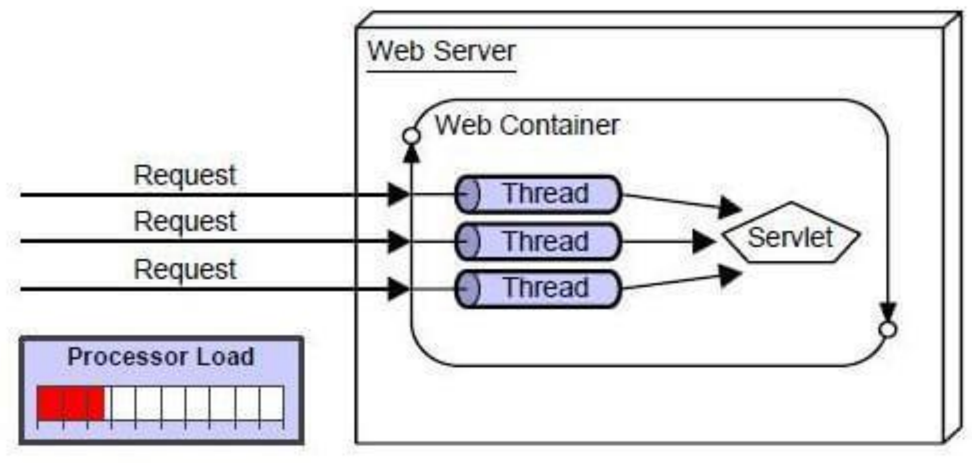


Figure 3: Use of Servlet in web server

The basic benefits of servlet are as follows:

- ✓ Better performance: because it creates a thread for each request not process.
- ✓ Portability: because it uses java language.
- ✓ Robust: Servlets are managed by JVM so we don't need to worry about memory leak, garbage collection etc.
- ✓ Secure: because it uses java language

Life Cycle of Servlets

Three methods are central to the life cycle of a servlet. These are **init( ), service( ),** and **destroy( )**. They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

First, when a user enters a Uniform Resource Locator (URL) to a Web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

Second, this HTTP request is received by the Web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

Third, the server invokes the init( ) method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

Fourth, the server invokes the service( ) method of the servlet. This method is called to process the HTTP request. It is possible for the servlet to read data that has been provided in the HTTP request.

It may also formulate an HTTP response for the client. The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The service( ) method is called for each HTTP request.

Finally, the server may decide to unload the servlet from its memory. The server calls the destroy( ) method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

A servlet life cycle can be defined as the entire process from its creation till the destruction

The following are the paths followed by a servlet

- ✓ The servlet is initialized by calling the **init()** method
- ✓ The servlet calls **service()** method to process a client's request.
- ✓ The servlet is terminated by calling the **destroy()** method
- ✓ Finally, servlet is garbage collected by the garbage collector of the JVM

init() method

The **init()** method is called only once when the servlet is first created, so, it is used for one-time Initializations. The **init()** method simply creates or loads some data that will be used throughout the life of the servlet.

```
public     void     init()     throws
ServletException {
// Initialization code...
}
```

service() method

The servlet container (i.e. web server) calls the **service()** method to handle requests coming from the client (browsers) and to write the formatted response back to the client. Each time the server receives a request for a servlet, the server spawns a new thread and calls **service().** The **service()** method checks the HTTP request type (GET, POST, PUT,DELETE, etc.) and calls *doGet, doPost, doPut, doDelete,* etc. methods as appropriate

```
public void service(
ServletRequest request,
ServletResponse response)
throws ServletException, IOException {
    // ...
    }
```

destroy() method

The **destroy()** method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to perform cleanup activities like close database connections, halt background threads, write cookie lists or hit counts to disk etc. After the **destroy()** method is called, the servlet object is marked for garbage collection

```
public void destroy() {
// Finalization code...
}
```

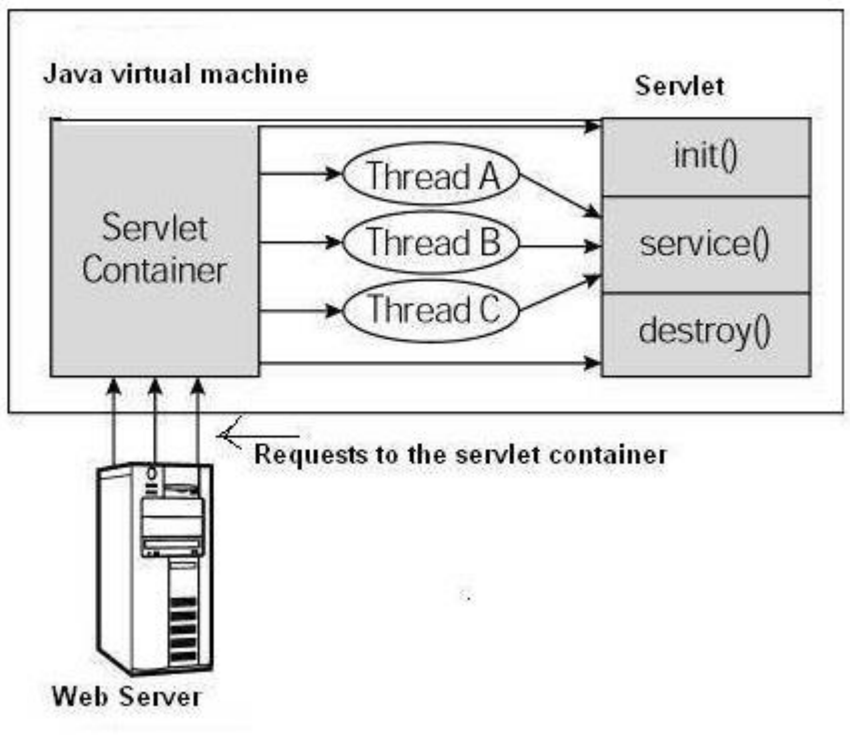


Figure 4: Lifecycle of servlet

According to above figure:

- First the HTTP requests coming to the server are delegated to the servlet container.
- The servlet container loads the servlet before invoking the service() method.
- Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.

For more clarity,

Web Server: It is also known as HTTP Server, it can handle HTTP Requests send by client and responds the request with an HTTP Response.

Web Container: Also known as Servlet Container and Servlet Engine. It is a part of Web Server that interacts with Servlets. This is the main component of Web Server that manages the life cycle of Servlets.

Servlet life cycle contains five steps: **1) Loading of Servlet 2) Creating instance of Servlet 3) Invoke init() once 4) Invoke service() repeatedly for each client request 5) Invoke destroy()**

Step 1: Loading of Servlet

When the web server (e.g. Apache Tomcat) starts up, the servlet container deploy and loads all the servlets.

Step 2: Creating instance of Servlet
Once all the Servlet classes loaded, the servlet container creates instances of each servlet class. Servlet container creates only once instance per servlet class and all the requests to the servlet are executed on the same servlet instance.

Step 3: Invoke init() method
Once all the servlet classes are instantiated, the init() method is invoked for each instantiated servlet. This method initializes the servlet. There are certain init parameters that you can specify in the deployment descriptor (web.xml) file. For example, if a servlet has value >=0 then its init() method is immediately invoked during web container startup.

You can specify the element in web.xml file like this:

```
<servlet>
 <servlet-name>MyServlet</servlet-name>
 <servlet-class>com.beginnersbook.MyServletDemo</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>
```

Now the init() method for corresponding servlet class **com.beginnersbook.MyServletDemo** would be invoked during web container startup.

Note: The init() method is called only once during the life cycle of servlet.

Step 4: Invoke service() method
Each time the web server receives a request for servlet, it spawns a new thread that calls service() method. If the servlet is GenericServlet then the request is served by the service() method itself, if the servlet is HttpServlet then service() method receives the request and dispatches it to the correct handler method based on the type of request.

For example if its a Get Request the service() method would dispatch the request to the doGet() method by calling the doGet() method with request parameters. Similarly the requests like Post, Head, Put etc. are dispatched to the corresponding handlers doPost(), doHead(), doPut() etc. by service() method of servlet.

Note: Unlike init() and destroy() that are called only once, the service() method can be called any number of times during servlet life cycle. As long as servlet is not destroyed, for each client request the service() method is invoked.

Out of all the 5 steps in life cycle, this is the only step that executes multiple times.

Step 5: Invoke destroy() method
When servlet container shuts down(this usually happens when we stop the web server), it unloads all the servlets and calls destroy() method for each initialized servlets.

Java Servlets Development Kit

The Java^{TM} Servlet Development Kit (JSDK), in conjunction with the JDK, contains all the pieces necessary for implementing servlets on Java-based web servers and any web servers or servlet engines that implement the **Java Servlet API**

The JSDK includes an implementation of the Java Servlet API, a servlet engine for running and testing servlets, the *javax.servlet* and *sun.servlet* package sources, API documentation for *javax.servlets*, and example code to help developers get started writing servlets

Java Servlet API

We need to use Servlet API to create servlets. There are two packages that you must remember while using API, the *javax.servlet* package that contains the classes to support generic servlet (protocol-independent servlet) and the *javax.servlet.http* package that contains classes to support http servlet.

Every Servlet must implement the **java.servlet.Servlet** interface, you can do it by extending one of the following two classes: **javax.servlet.GenericServlet** or **javax.servlet.http.HttpServlet**. The first one is for protocol independent Servlet and the second one for http Servlet.

- If you are creating a Generic Servlet then you must extend **javax.servlet.GenericServlet** class. GenericServlet class has an abstract *service()* method. Which means the subclass of **GenericServlet** should always override the *service()* method.
- If you creating Http Servlet you must extend **javax.servlet.http.HttpServlet** class, which is an abstract class. Unlike Generic Servlet, the HTTP Servlet doesn't override the *service()* method. Instead it overrides one or more of the following methods. It must override at least one method from the list below:
    - **doGet() –** This method is called by servlet service method to handle the HTTP GET request from client. The Get method is used for getting information from the server
    - **doPost() –** Used for posting information to the Server
    - **doPut()** – This method is similar to doPost method but unlike doPost method where we send information to the server, this method sends file to the server, this is similar to the FTP operation from client to server
    - **doDelete()** – allows a client to delete a document, webpage or information from the server

- **init() and destroy()** – Used for managing resources that are held for the life of the servlet
- **getServletInfo()** – Returns information about the servlet, such as author, version, and copyright.

Servlet vs GenericServlet vs HttpServlet

Servlets are platform-independent server-side components, being written in Java. Before going for differences, first let us see how the three Servlet, GenericServlet, HttpServlet are related, their signatures and also at the end similarities.

To write a servlet, everyone goes to extend the abstract class HttpServlet, like Frame is required to extend to create a frame.

Following figure shows the hierarchy of Servlet vs GenericServlet vs HttpServlet and to know from where HttpServlet comes.

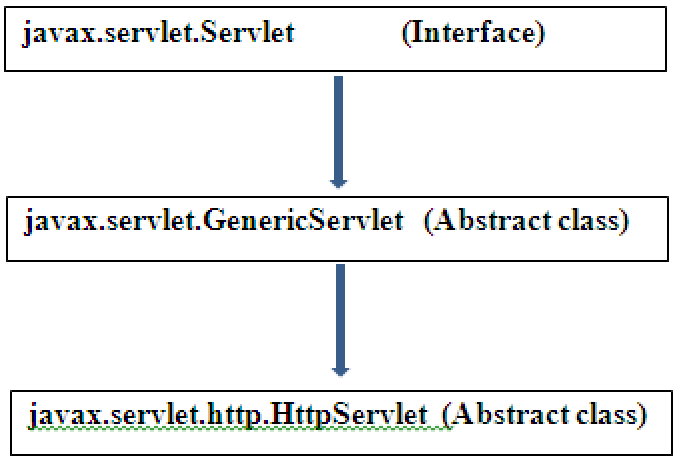

javax.servlet.Servlet (Interface)
javax.servlet.GenericServlet (Abstract class)
javax.servlet.http.HttpServlet (Abstract class)

Figure 5: Servlet vs GenericServlet vs HttpServlet

Observe the hierarchy and understand the relationship between the three (involved in multilevel inheritance). With the observation, a conclusion can be arrived, to write a Servlet three ways exist.

a) by implementing Servlet (it is interface)
b) by extending GenericServlet (it is abstract class)
c) by extending HttpServlet (it is abstract class)

The minus point of the first way is, all the 5 abstract methods of the interface Servlet should be overridden even though Programmer is not interested in all (like the interface WindowListener to close the frame). A smart approach is inheriting GenericServlet (like using WindowAdapter) and overriding its only one abstract method service(). It is enough to the programmer to override only this method. It is a callback method (called implicitly). Still better way is extending HttpServlet and need not to override any methods as HttpServlet contains no abstract methods. Even though the HttpServlet does not contain any abstract methods, it is declared as abstract class by the Designers to not to allow the Programmer to create an object directly because a Servlet object is created by the system (here system is Servlet Container).

Servlet interface

It is the super interface for the remaining two – GenericServlet and HttpServlet. It contains 5 abstract methods and all inherited by GenericServlet and HttpServlet. Programmers implement Servlet interface who would like to develop their own container

GenericServlet

It is the immediate subclass of Servlet interface. In this class, only one abstract method service() exist. Other 4 abstract methods of Servlet interface are given implementation (given body). Anyone who extends this GenericServlet should override service() method. It was used by the Programmers when the Web was not standardized to HTTP protocol. It is protocol independent; it can be used with any protocol, say, SMTP, FTP, CGI including HTTP etc.

Signature:

public abstract class GenericServlet extends java.lang.Object implements Servlet, ServletConfig, java.io.Serializable

HttpServlet

When HTTP protocol was developed by W3C people to suit more Web requirements, the Servlet designers introduced HttpServlet to suit more for HTTP protocol. HttpServlet is protocol dependent and used specific to HTTP protocol only.

The immediate super class of HttpServlet is GenericServlet. HttpServlet overrides the service() method of GenericServlet. HttpServlet is abstract class but without any abstract methods.

With HttpServlet extension, service() method can be replaced by doGet() or doPost() with the same parameters of service() method.

Signature:

public abstract class HttpServlet extends GenericServlet implements java.io.Serializable

Being subclass of GenericServlet, the HttpServlet inherits all the properties (methods) of GenericServlet. So, if you extend HttpServlet, you can get the functionality of both.

Differences

GENERICSERVLET	**HTTPSERVLET**
Can be used with any protocol (means, can handle any protocol). Protocol independent.	Should be used with HTTP protocol only (can handle HTTP specific protocols) . Protocol dependent.
All methods are concrete except service() method. service() method is abstract method.	All methods are concrete (non-abstract). service() is non-abstract method.
service() should be overridden being abstract in super interface.	service() method need not be overridden.
It is a must to use service() method as it is a callback method.	Being service() is non-abstract, it can be replaced by doGet() or doPost() methods.
Extends Object and implements interfaces Servlet, ServletConfig and Serializable.	Extends GenericServlet and implements interface Serializable
Direct subclass of Servet interface.	Direct subclass of GenericServlet.
Defined javax.servlet package.	Defined javax.servlet.http package.
All the classes and interfaces belonging to javax.servlet package are protocol independent.	All the classes and interfaces present in javax.servlet.http package are protocol dependent (specific to HTTP).
Not used now-a-days.	Used always.

Similarities :

1. One common feature is both the classes are abstract classes.
2. Used with Servlets only.

Creating, Compiling and running servlets

Create a servlet in a web application (HelloServlet)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet {
private String message;
public void init() throws ServletException {
message = "Hello World";
}
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<h1>" + message + "</h1>");
}
public void destroy() { // ... }
}
```

- Right click on the project in NetBeans and click on Build menu item
- To run the servlet

  Right click on the project or servlet file and click on Run

  Or

. Start GlassFish server manually

  asadmin start-domain –verbose

- Copy the .war file in the dist subfolder to

  glassfish/domains/domain1/autodeploy folder

- Open a browser and run servlet using the URL

  http://localhost:8080/HelloServlet/HelloWorld

Note:

On Windows, GlassFish server generally starts on port 4848, so use

http://localhost:4848/HelloServlet/HelloWorld

You can stop the server using asadmin stop-domain domain1.

(port numbers may vary…)

Deployment Descriptor web.xml in Servlets

What is Deployment?

Copying the **.class** file of the Servlet from the current directory to the classes folder of **Tomcat** (or any Web server) is known as **deployment**. When deployed, Tomcat is ready to load and execute the Servlet, at any time, at the client request.

What is Deployment Descriptor?

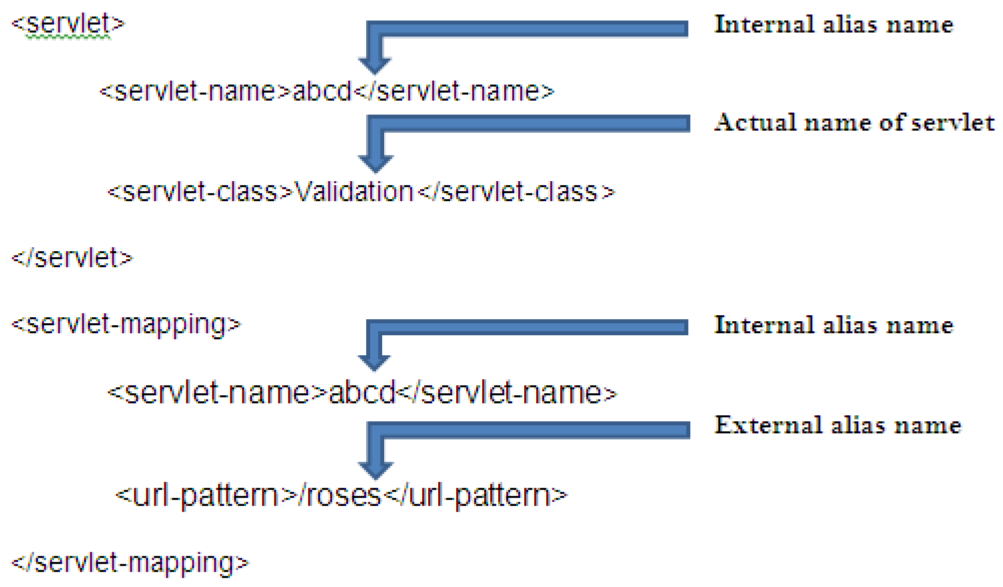
As the name indicates, the deployment descriptor describes the deployment information (or Web Information) of a Servlet. The **deployment descriptor** is **an XML file known as web.xml**. XML is the easiest way to give the information to a server, just writing in between the tags, instead of writing in a text file or RDBMS file. The name and tags of web.xml are Servlet API specifications.

What information can be stored with deployment descriptor?

The following activities can be done by the programmer in **web.xml** file.

1. **Mapping alias name with the actual Servlet name**

First and foremost is the alias name to the Servlet. Never a client is given the actual name of the Servlet. Always an alias name is given just for security (avoid hacking). The alias name is given in the following XML tags.



```
<servlet>
        <servlet-name>abcd</servlet-name>
         <servlet-class>Validation</servlet-class>
</servlet>
<servlet-mapping>
         <servlet-name>abcd</servlet-name>
          <url-pattern>/roses</url-pattern>
</servlet-mapping>
```

The Servlet comes with two **alias** names, **internal and external**. The internal name is used by the Tomcat and the external name is given (to be written in <FORM> tag of HTML file) to the client to invoke the Servlet on the server. That is, there exists alias to alias. All this is for security. Observe, the names are given in two different XML tags, in the web.xml file, to make it difficult for hacking (for more security in EJB(Enterprise Java Beans)), two alias are given in two different XML files).

To invoke the Validation Servlet, the client calls the server with the name roses. When roses call reaches the server, the Tomcat server opens the web.xml file to check the deployment particulars. Searches such a **<servlet-mapping>** tag that matches roses. roses is exchanged with abcd. Then, searches such a **<servlet>** tag that matches abcd and exchanges with Validation. Now the server, loads Validation Servlet, executes and sends the output of execution as response to client.

1. **To write Initialization Parameters**

Intialization parameteres are read by the Servlet from web.xml file. Programmer can write code to be used for initialization. An example code is given below

```
<init-param>
  <param-name>trainer</param-name>
  <param-value>S. Nageswara Rao</param-value>
</init-param>
```

2. **To write tag libraries**

   (this is mostly used in frameworks like Struts etc)

Following are a few important XML elements in web.xml.

<context-param>, <filter-mapping>, <taglib> and <mime-mapping> etc.

Note 1: When a Servlet code is modified and copied to classes folder, each time it is required to restart the Tomcat server.

Note 2: For every Servlet, there must be an entry in web.xml file with an alias name.

Reading the Servlet Parameters

Servlets handles form data parsing automatically using the following methods depending on the situation:

- `getParameter()`
  we call request.getParameter() method to get the value of a form parameter.
- `getParameterValues()`
  Call this method if the parameter appears more than once and returns multiple values, for example checkbox.
- `getParameterNames()`
  Call this method if you want a complete list of all parameters in the current request

  Example:

Hello.html

```
<html>
<body>
<form action="HelloForm" method="GET">
First Name: <input type="text"
name="first_name">
<br />
Last Name: <input type="text"
name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
    </html>
```

In web.xml

```
<servlet>
<servlet-name>ServletName</servlet-name>
<servlet-class>com.example.ServletDemo</servlet-
class>
<init-param>
<param-name>email</param-name>
<param-value>admin@email.com</param-value>
</init-param>
</servlet>
<servlet-mapping>
<servlet-name>ServletName</servlet-name>
<url-pattern>/Demo</url-pattern>
</servlet-mapping>
```

Servlet code

```
        public void doGet(HttpServletRequest
        request,
        HttpServletResponse response)
        throws IOException {
        PrintWriter           out              =
        response.getWriter();
        out.println(getServletConfig().getIn
        itParameter(
        "email")
        );
        }
```

The javax.servlet.http Package

The javax.servlet.http package supports the development of servlets that use the HTTP protocol.The classes in this package extend the basic servlet functionality to support various HTTP specific features, including request and response headers,different request methods, and cookies.

Interfaces

- HttpServletRequest
- HttpServletResponse
- HttpSession
- HttpSessionListener
- HttpSessionAttributeListener
- HttpSessionBindingListener
- HttpSessionActivationListener
- HttpSessionContext (deprecated now)

Classes

- HttpServlet
- Cookie
- HttpServletRequestWrapper
- HttpServletResponseWrapper
- HttpSessionEvent
- HttpSessionBindingEvent
- HttpUtils (deprecated now)

Handling HTTP Request and Response

The HttpServlet class provides methods that handle various types of HTTP requests.

A servlet developer typically overrides one of these methods calls **doGet(), doPost(), doPut(),doDelete()** etc.

EXAMPLE:

Following example takes a number from html form and generates and displays the table of that number.

- **In NetBeans IDE, Create project-> Java Web ->Web Application-> MyWebApp(project name)**
- **Server: Apache Tomcat 8.0.27.0**
- **Java EE version : Java EE 7 Web**

MyWebApp->WebPages->[index.html, generateTable.html]

index.html

```
<html>
        <head>
          <title>MyWebApps</title>
          <meta charset="UTF-8">
          <meta name="viewport" content="width=device-width, initial-scale=1.0">
        </head>
        <body>
          <div><h1>Here are  My Web Aplications!</h1>
            <h2><a href="generateTable.html">Click Here</a> to see my App1</h2>
          </div>
        </body>
      </html>
```

generateTable.html

```
<html>
  <head>
    <title>App1</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <div><h1>Welcome to my App1</h1>
      <h3>This App will generate the table of a given Number </h3>
      <form action="GenerateTableServlet" method="get">
        <table>
          <tr>
            <td>Enter a number: </td>
            <td><input type="text" name="number"></td></tr>
            <tr> <td> <input type="Submit" value="GenerateTable"></td></tr>


        </table>

      </form>

    </div>

  </body>

</html>
```

MyWebApp->Source Packages->(create new package with name 'mywebapp')->inside mywebapp package create a new servlet with name 'GenerateTableServlet'.

[tick on 'add information to deployment descriptor (web.xml)]

<u>GenerateTableServlet.java</u>

```
package mywebapp;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;


/**

 *

 * @author BIPIN

 */

public class GenerateTableServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        int a = Integer.parseInt(request.getParameter("number"));
        int res;
        out.println("<div align ='center'>");
        for(int i= 1; i<=10;i++){
            res=a*i;
        out.println("<h3>"+a+" * "+i+" = "+ res + "</h3>");
        }
        out.println("</div>");
    }
}
```

MyWeApp->Web Pages->WEB-INF->web.xml

<u>web.xml</u>

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
    <servlet>
        <servlet-name>GenerateTableServlet</servlet-name>
        <servlet-class>mywebapp.GenerateTableServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>GenerateTableServlet</servlet-name>
        <url-pattern>/GenerateTableServlet</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

MyWeApp->Web Pages->META-INF->context.xml

context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/MyWebApp"/>
```

Services->Servers->Apache Tomcat [start]

Go to web browser, type url -> http://localhost:8084/MyWebApp/

Or simple run file in NetBeans

Using Cookies

- Cookies are text files stored on the client computer and they are kept for various information tracking purpose. There are three steps involved in identifying users through cookies
- Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user

A cookie is a small piece of information that is persisted between the multiple client requests. A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

How Cookie works

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

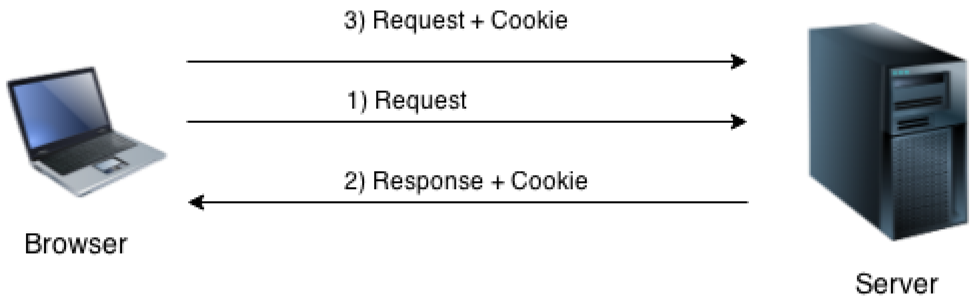


Figure 6:Cookie

Types of Cookie

There are 2 types of cookies in servlets.

1. **Non-persistent cookie:** It is valid for single session only. It is removed each time when user closes the browser.
2. **Persistent cookie:** It is valid for multiple session. It is not removed each time when user closes the browser. It is removed only if user logout or signout.

Advantage of Cookies

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

Disadvantage of Cookies

- It will not work if cookie is disabled from the browser.
- Only textual information can be set in Cookie object.

Setting Cookies with Servlet

Setting cookies with servlet involves three steps −

(1) Creating a Cookie object − You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

```
Cookie cookie = new Cookie("key","value");
```

Keep in mind, neither the name nor the value should contain white space or any of the following characters −

```
[ ] ( ) = , " / ? @ : ;
```

(2) Setting the maximum age − You use setMaxAge to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

```
cookie.setMaxAge(60 * 60 * 24);
```

(3) **Sending the Cookie into the HTTP response headers** − You use response.addCookie to add cookies in the HTTP response header as follows−

```
response.addCookie(cookie);
```

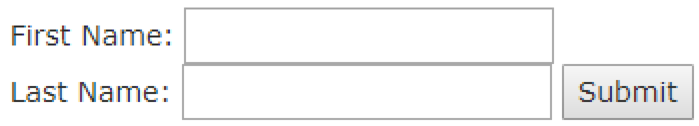
Example:

```
// Import required java libraries

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;


// Extend HttpServlet class

public class HelloForm extends HttpServlet {


   public void doGet(HttpServletRequest request, HttpServletResponse response)

      throws ServletException, IOException {


      // Create cookies for first and last names.

      Cookie firstName = new Cookie("first_name",
request.getParameter("first_name"));

      Cookie lastName = new Cookie("last_name", request.getParameter("last_name"));


      // Set expiry date after 24 Hrs for both the cookies.

      firstName.setMaxAge(60*60*24);

      lastName.setMaxAge(60*60*24);


      // Add both the cookies in the response header.

      response.addCookie( firstName );

      response.addCookie( lastName );


      // Set response content type

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      String title = "Setting Cookies Example";

      String docType =

         "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";


      out.println(docType +

         "<html>\n" +

            "<head>

               <title>" + title + "</title>

            </head>\n" +


            "<body bgcolor = \"#f0f0f0\">\n" +

               "<h1 align = \"center\">" + title + "</h1>\n" +

               "<ul>\n" +

                  "  <li><b>First Name</b>: "

                  + request.getParameter("first_name") + "\n" +

                  "  <li><b>Last Name</b>: "

                  + request.getParameter("last_name") + "\n" +

               "</ul>\n" +

            "</body>

         </html>"

      );

   }

}
```

Compile the above servlet HelloForm and create appropriate entry in web.xml file and finally try following HTML page to call servlet.

```
<html>

   <body>

      <form action = "HelloForm" method = "GET">

         First Name: <input type = "text" name = "first_name">

         <br />

         Last Name: <input type = "text" name = "last_name" />

         <input type = "submit" value = "Submit" />

      </form>

   </body>

</html>
```

Keep above HTML content in a file Hello.htm and put it in <Tomcat-installationdirectory>/webapps/ROOT directory. When you would access http://localhost:8080/Hello.htm, here is the actual output of the above form.



Try to enter First Name and Last Name and then click submit button. This would display first name and last name on your screen and same time it would set two cookies firstName and lastName which would be passed back to the server when next time you would press Submit button.

Next section would explain you how you would access these cookies back in your web application.

Reading Cookies with Servlet

To read cookies, you need to create an array of ***javax.servlet.http.Cookie*** objects by calling the **getCookies()** method of ***HttpServletRequest.*** Then cycle through the array, and use **getName()** and **getValue()** methods to access each cookie and associated value.

Example

Let us read cookies which we have set in previous example −

```
// Import required java libraries

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;


// Extend HttpServlet class

public class ReadCookies extends HttpServlet {


   public void doGet(HttpServletRequest request, HttpServletResponse response)

      throws ServletException, IOException {


      Cookie cookie = null;

      Cookie[] cookies = null;


      // Get an array of Cookies associated with this domain

      cookies = request.getCookies();


      // Set response content type

      response.setContentType("text/html");


      PrintWriter out = response.getWriter();

      String title = "Reading Cookies Example";

      String docType =

         "<!doctype html public \"-//w3c//dtd html 4.0 " +

         "transitional//en\">\n";


      out.println(docType +

         "<html>\n" +
         "<head><title>" + title + "</title></head>\n" +
         "<body bgcolor = \"#f0f0f0\">\n" );

      if( cookies != null ) {
         out.println("<h2> Found Cookies Name and Value</h2>");

         for (int i = 0; i < cookies.length; i++) {
            cookie = cookies[i];
            out.print("Name : " + cookie.getName( ) + ",  ");
            out.print("Value: " + cookie.getValue( ) + " <br/>");
         }
      } else {
         out.println("<h2>No cookies founds</h2>");
      }
      out.println("</body>");
      out.println("</html>");
   }
}
```

Compile above servlet ReadCookies and create appropriate entry in web.xml file. If you would have set first_name cookie as "John" and last_name cookie as "Player" then running http://localhost:8080/ReadCookies would display the following result −

Found Cookies Name and Value

```
Name : first_name, Value: John
Name : last_name,  Value: Player
```

Delete Cookies with Servlet

To delete cookies is very simple. If you want to delete a cookie then you simply need to follow up following three steps −

- Read an already existing cookie and store it in Cookie object.

- Set cookie age as zero using **setMaxAge()** method to delete an existing cookie
- Add this cookie back into response header.

Example

The following example would delete and existing cookie named "first_name" and when you would run ReadCookies servlet next time it would return null value for first_name.

```
// Import required java libraries

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;


// Extend HttpServlet class

public class DeleteCookies extends HttpServlet {


   public void doGet(HttpServletRequest request, HttpServletResponse response)

      throws ServletException, IOException {


      Cookie cookie = null;

      Cookie[] cookies = null;


      // Get an array of Cookies associated with this domain

      cookies = request.getCookies();


      // Set response content type

      response.setContentType("text/html");


      PrintWriter out = response.getWriter();

      String title = "Delete Cookies Example";

      String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";

      out.println(docType +
         "<html>\n" +
         "<head><title>" + title + "</title></head>\n" +
         "<body bgcolor = \"#f0f0f0\">\n" );

      if( cookies != null ) {
         out.println("<h2> Cookies Name and Value</h2>");

         for (int i = 0; i < cookies.length; i++) {
            cookie = cookies[i];

            if((cookie.getName( )).compareTo("first_name") == 0 ) {
               cookie.setMaxAge(0);
               response.addCookie(cookie);
               out.print("Deleted cookie : " + cookie.getName( ) + "<br/>");
            }
            out.print("Name : " + cookie.getName( ) + ",  ");
            out.print("Value: " + cookie.getValue( )+" <br/>");
         }
      } else {
         out.println("<h2>No cookies founds</h2>");
      }
      out.println("</body>");
      out.println("</html>");
   }
```

```
}
```

Compile above servlet DeleteCookies and create appropriate entry in web.xml file. Now running http://localhost:8080/DeleteCookies would display the following result −

Cookies Name and Value

```
Deleted cookie : first_name
```

```
Name : first_name, Value: John
```

```
Name : last_name,  Value: Player
```

Now try to run http://localhost:8080/ReadCookies and it would display only one cookie as follows −

Found Cookies Name and Value

```
Name : last_name,  Value: Player
```

You can also delete your cookies in browser manually.

Servlets - Session Tracking

Session simply means a particular interval of time.

Session Tracking is a way to maintain state (data) of a user. It is also known as session management in servlet.

HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.

HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:

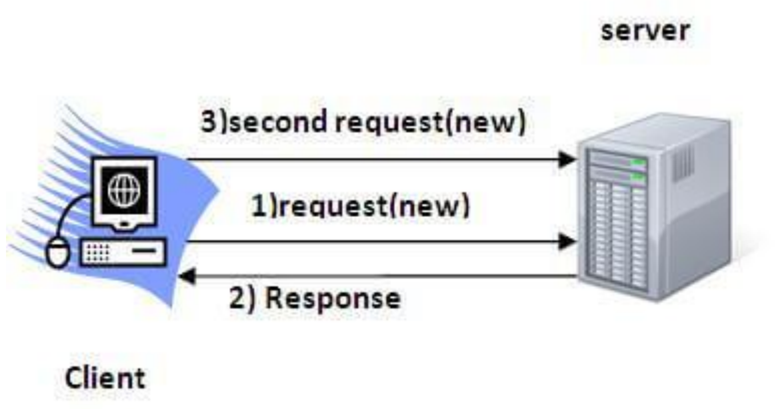


Figure 7: Figure to demonstrate HTTP is a stateless protocol

Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

The shopping cart software is classic example, a client can selected the items from multiply actions in his virtual box. Other examples include sites that use online communication portals, or database managing.

Great number of web-oriented application requires that application has to keep track of the clients performing actions, plain http doesn't provide that, and thus can't be supported without an additional API. To support the software that needs keep track of the state, Java Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

There are following ways to maintain session between web client and web server (i.e. State can be maintained indirectly) using –

1. **Cookies**
2. **Hidden Form Fields**
3. **URL Rewriting**
4. **HttpSession Object**

<u>Cookies</u>

A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.

This may not be an effective way because many time browser does not support a cookie.

Hidden Form Fields

A web server can send a hidden HTML form field along with a unique session ID as follows −

```
<input type = "hidden" name = "sessionid" value = "12345">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. Each time when web browser sends request back, then session_id value can be used to keep the track of different web browsers.

This could be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

URL Rewriting

You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session.

For example, with http://tutorialspoint.com/file.htm;sessionid = 12345, the session identifier is attached as sessionid = 12345 which can be accessed at the web server to identify the client.

URL rewriting is a better way to maintain sessions and it works even when browsers don't support cookies. The drawback of URL re-writing is that you would have to generate every URL dynamically to assign a session ID, even in case of a simple static HTML page.

The HttpSession Object

Apart from the above mentioned three ways, servlet provides **HttpSession** Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user.

You would get **HttpSession** object by calling the public method **getSession()**of **HttpServletRequest**, as below −

```
HttpSession session = request.getSession();
```

You need to call **request.getSession()** before you send any document content to the client. Here is a summary of the important methods available through **HttpSession** object −

Sr.No.	Method & Description
1	**public Object getAttribute(String name)** This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	**public Enumeration getAttributeNames()** This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3	**public long getCreationTime()** This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4	**public String getId()** This method returns a string containing the unique identifier assigned to this session.
5	**public long getLastAccessedTime()** This method returns the last accessed time of the session, in the format of milliseconds since midnight January 1, 1970 GMT
6	**public int getMaxInactiveInterval()** This method returns the maximum time interval (seconds), that the servlet container will keep the session open between client accesses.
7	**public void invalidate()** This method invalidates this session and unbinds any objects bound to it.
8	**public boolean isNew(** This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
9	**public void removeAttribute(String name)**

	This method removes the object bound with the specified name from this session.
10	**public void setAttribute(String name, Object value)** This method binds an object to this session, using the name specified.
11	**public void setMaxInactiveInterval(int interval)** This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

Deleting Session Data

When you are done with a user's session data, you have several options −

- **Remove a particular attribute** − You can call public void *removeAttribute(String name)* method to delete the value associated with a particular key.
- **Delete the whole session** − You can call *public void invalidate()*method to discard an entire session.
- **Setting Session timeout** − You can call public void *setMaxInactiveInterval(int interval)* method to set the timeout for a session individually.
- **Log the user out** − The servers that support servlets 2.4, you can call logout to log the client out of the Web server and invalidate all sessions belonging to all the users.
- **web.xml Configuration** − If you are using Tomcat, apart from the above mentioned methods, you can configure session time out in web.xml file as follows.

```
<session-config>

   <session-timeout>15</session-timeout>

</session-config>
```

The timeout is expressed as minutes, and overrides the default timeout which is 30 minutes in Tomcat.

The *getMaxInactiveInterval( )* method in a servlet returns the timeout period for that session in seconds. So if your session is configured in web.xml for 15 minutes, *getMaxInactiveInterval( )* returns 900.

Session Tracking Example

This example describes how to use the HttpSession object to find out the creation time and the last-accessed time for a session. We would associate a new session with the request if one does not already exist.

```
// Import required java libraries

import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

import java.util.*;


// Extend HttpServlet class

public class SessionTrack extends HttpServlet {


   public void doGet(HttpServletRequest request, HttpServletResponse response)

      throws ServletException, IOException {


      // Create a session object if it is already not  created.

      HttpSession session = request.getSession(true);


      // Get session creation time.

      Date createTime = new Date(session.getCreationTime());


      // Get last access time of this web page.

      Date lastAccessTime = new Date(session.getLastAccessedTime());


      String title = "Welcome Back to my website";

      Integer visitCount = new Integer(0);

      String visitCountKey = new String("visitCount");
      String userIDKey = new String("userID");
      String userID = new String("ABCD");

      // Check if this is new comer on your web page.
      if (session.isNew()) {
         title = "Welcome to my website";
         session.setAttribute(userIDKey, userID);
      } else {
         visitCount = (Integer)session.getAttribute(visitCountKey);
         visitCount = visitCount + 1;
         userID = (String)session.getAttribute(userIDKey);
      }
      session.setAttribute(visitCountKey,  visitCount);

      // Set response content type
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " +
         "transitional//en\">\n";

      out.println(docType +
         "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +

            "<body bgcolor = \"#f0f0f0\">\n" +
               "<h1 align = \"center\">" + title + "</h1>\n" +
               "<h2 align = \"center\">Session Infomation</h2>\n" +
               "<table border = \"1\" align = \"center\">\n" +

                  "<tr bgcolor = \"#949494\">\n" +
                     "  <th>Session info</th><th>value</th>
                  </tr>\n" +

                  "<tr>\n" +
                     "  <td>id</td>\n" +
                     "  <td>" + session.getId() + "</td>
                  </tr>\n" +

                  "<tr>\n" +
                     "  <td>Creation Time</td>\n" +
                     "  <td>" + createTime + "  </td>
                  </tr>\n" +

                  "<tr>\n" +
                     "  <td>Time of Last Access</td>\n" +
                     "  <td>" + lastAccessTime + "  </td>
                  </tr>\n" +

                  "<tr>\n" +
                     "  <td>User ID</td>\n" +
                     "  <td>" + userID + "  </td>
                  </tr>\n" +


                  "<tr>\n" +
                     "  <td>Number of visits</td>\n" +
                     "  <td>" + visitCount + "</td>
                  </tr>\n" +
               "</table>\n" +
            "</body>
         </html>"
      );
   }
}
```

Compile the above servlet SessionTrack and create appropriate entry in web.xml file. Now running http://localhost:8080/SessionTrack would display the following result when you would run for the first time −

Welcome to my website

Session Infomation

Session info	**value**
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	0

Now try to run the same servlet for second time, it would display following result.

Welcome Back to my website

Session Infomation

info type	**value**
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	1

Java Server Pages (JSP)

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. The JSP tags which allow java code to be included into it are **<% ----java code----%>.**

It can consist of either HTML or XML (combination of both is also possible) with JSP actions and commands.

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.

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 enables us to write HTML pages containing tags, inside which we can include powerful Java programs. Using JSP, one can easily separate Presentation and Business logic as a web designer can design and update JSP pages creating the presentation layer and java developer can write server side complex computational code without concerning the web design. And both the layers can easily interact over HTTP requests.

- It can be used as HTML page, which can be used in forms and registration pages with the dynamic content into it.
- Dynamic content includes some fields like dropdown, checkboxes, etc. whose value will be fetched from the database.
- This can also be used to access JavaBeans objects.
- We can share information across pages using request and response objects.
- JSP can be used for separation of the view layer with the business logic in the web application.

Why JSP?

- In Java server pages JSP, the execution is much faster compared to other dynamic languages.
- It is much better than Common Gateway Interface (CGI).
- Java server pages JSP are always compiled before it’s processed by the server as it reduces the effort of the server to create process.
- Java server pages JSP are built over Java Servlets API. Hence, it has access to all Java APIs, even it has access to JNDI, JDBC EJB and other components of java.
- JSP are used in MVC architecture (which will be covered in MVC architecture topic) as view layer.
- The request is processed by a view layer which is JSP and then to servlet layer which is java servlet and then finally to a model layer class which interacts with the database.
- JSP is an important part of Java EE, which is a platform for enterprise level applications

Advantages of JSP

- The advantage of JSP is that the programming language used is JAVA, which is a dynamic language and easily portable to other operating systems.
- It is very much convenient to modify the regular HTML. We can write the servlet code into the JSP.
- It is only intended for simple inclusions which can use form data and make connections.
- JSP can also include the database connections into it. It can contain all type of java objects.
- It is very easy to maintain
- Performance and scalability of JSP are very good because JSP allows embedding of dynamic elements in HTML pages.
- As it is built on Java technology, hence it is platform independent and not depending on any operating systems.
- Also, it includes the feature of multithreading of java into it.
- We can also make use of exception handling of java into JSP.
- It enables to separate presentation layer with the business logic layer in the web application.
- It is easy for developers to show as well as process the information.

Advantages of JSP over Servlets

We can generate HTML response from servlets also but the process is cumbersome and error prone, when it comes to writing a complex HTML response, writing in a servlet will be a nightmare. JSP helps in this situation and provide us flexibility to write normal HTML page and include our java code only where it's required.

JSP provides additional features such as tag libraries, expression language, custom tags that helps in faster development of user views.

JSP pages are easy to deploy, we just need to replace the modified page in the server and container takes care of the deployment. For servlets, we need to recompile and deploy whole project again.

Actually Servlet and JSPs compliment each other. We should use Servlet as server side controller and to communicate with model classes whereas JSPs should be used for presentation layer.

- JSP provides an easier way to code dynamic web pages.
- JSP does not require additional files like, java class files, web.xml etc

- Any change in the JSP code is handled by Web Container (Application server like tomcat), and doesn't require re-compilation.
- JSP pages can be directly accessed, and web.xml mapping is not required like in servlets.

Servlets vs JSP

Java Servlets are programs that run on a Web or Application server and act as a middle layer between a request coming from a Web browser or other HTTP client and databases or applications on the HTTP server.

Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

JavaServer Pages (JSP) is a technology for developing web pages that support dynamic content which helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>.

JSP are webpages similar to aspx/php pages which run at server side. Javascript or Html code in JSP pages runs at client side.

Key Differences :

- Servlet is **html in java** whereas JSP is **java in html**.
- Servlets run faster compared to JSP
- JSP can be compiled into Java Servlets
- It's easier to code in JSP than in Java Servlets
- JSP is a webpage scripting language that can generate dynamic content while Servlets are Java programs that are already compiled which also creates dynamic web content
- In MVC, jsp acts as a view and servlet acts as a controller.
- JSP are generally preferred when there is not much processing of data required. But servlets are best for use when there is more processing and manipulation involved.
- The advantage of JSP programming over servlets is that we can build custom tags which can directly call Java beans. There is no such facility in servlets.
- We can achieve functionality of JSP at client side by running JavaScript at client side. There are no such methods for servlets.

- A servlet is like any other Java class. You put HTML into print statements like you use System.out or how JavaScript uses document.write. A JSP technically gets converted to a servlet but it looks more like PHP files where you embed the Java into HTML.

JSP is a webpage scripting language that can generate dynamic content while Servlets are Java programs that are already compiled which also creates dynamic web content Servlets run faster compared to JSP.JSP can be compiled into Java Servlets. It is easier to code in JSP than in Java JSP and Java Servlets are usually used in conjunction nowadays.

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

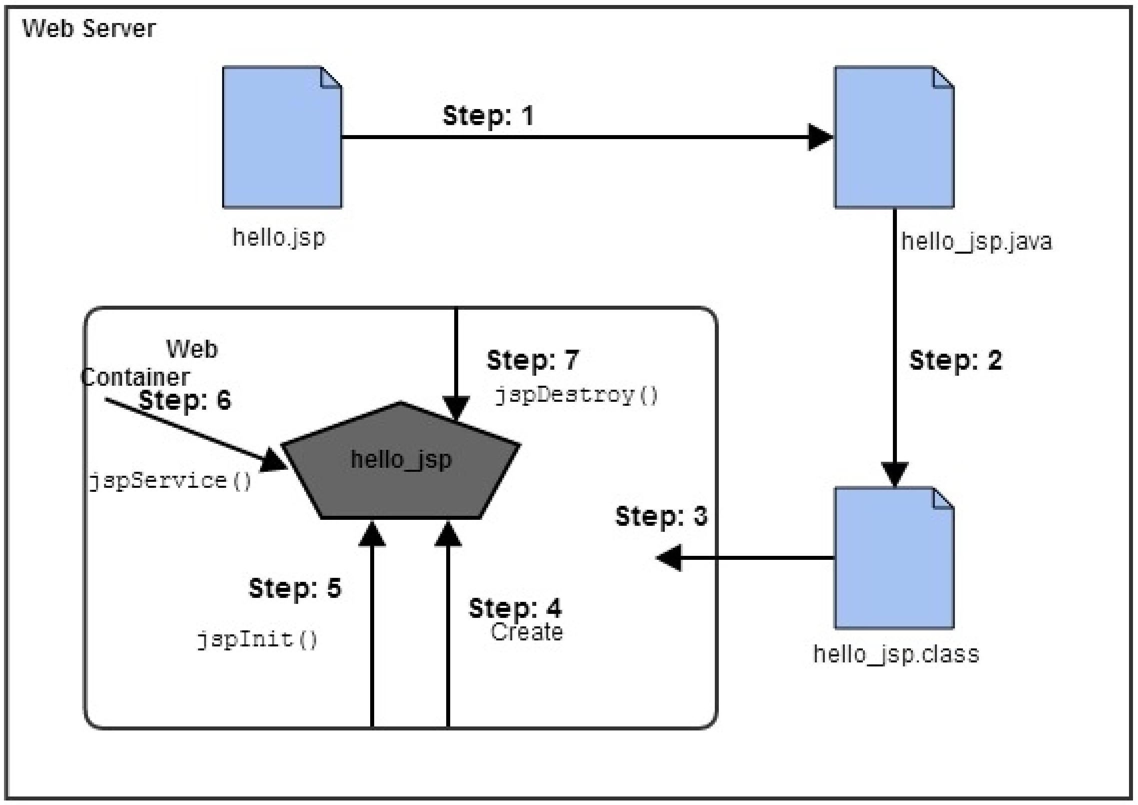


Figure 8: JSP Life Cycle

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.setContenType("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 lifecycle methods:

- ✓ **jspInit()** declared in JspPage interface. This method is called only once in JSP lifecycle to initialize config params.

- ✓ **_jspService(HttpServletRequest request, HttpServletResponse response)** declared in HttpJspPage interface and response for handling client requests.
- ✓ **jspDestroy()** declared in JspPage interface to unload the JSP from memory.

JSP Architecture

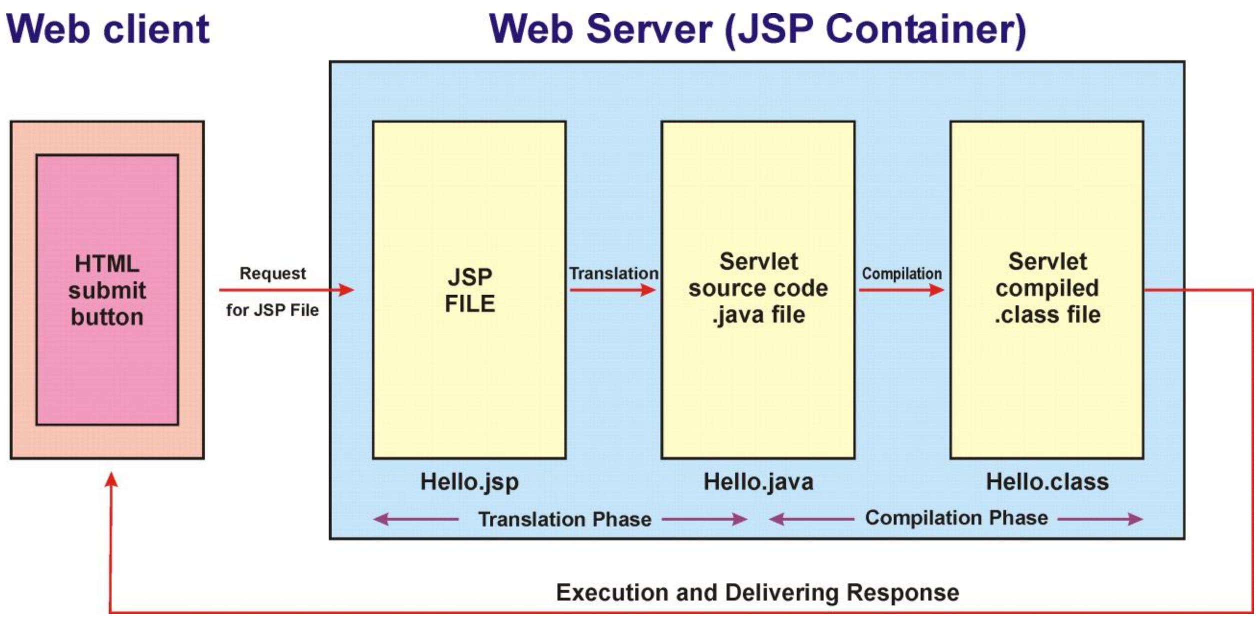


Figure 9: Architecture of JSP

Following steps includes the JSP Architecture as shown in the above figure.

1. Web client sends request to Web server for a JSP page (extension .jsp).
2. As per the request, the Web server (here after called as JSP container) loads the page.
3. JSP container converts (or translates) the JSP file into Servlet source code file (with extension .java). This is known as translation.
4. The translated .java file of Servlet is compiled that results to Servlet .class file. This is known as compilation.
5. The .class file of Servlet is executed and output of execution is sent to the client as response.

The JSP container converts the JSP file into a Servlet source file with extension .java. It is simply as equivalent you write a Servlet file. That is, instead of you write a Servlet file, container writes (as per the source code of JSP file). That means, a JSP file is converted into a Servlet internally. This is known as translation and translation phase. Some people call it as parsing. To do translation, some tools use PageCompileServlet. Of course, it is entirely IDE tools or Container software dependent. Resin JSP container
uses JspPrecompileListener and JspCompiler. Ecplise puts everything in a WAR file.

The JSP container software compiles the Servlet source code (.java file) into a Servlet .class file. This is known as JSP compilation.

The .class file of Servlet is executed by the JSP container and the output of execution is sent to client as response. If the same JSP file is not used again, the .class file is deleted. If used again and often, the .class file is retained by the container for further usage without the need of second time translation and compilation. This is to increase the performance.

JSP Access Model

The early JSP specifications advocated two philosophical approaches, popularly known as Model 1 and Model 2 architectures, for applying JSP technology. These approaches differ essentially in the location at which the bulk of the request processing was performed, and offer a useful paradigm for building applications using JSP technology.

Model 1 Architecture

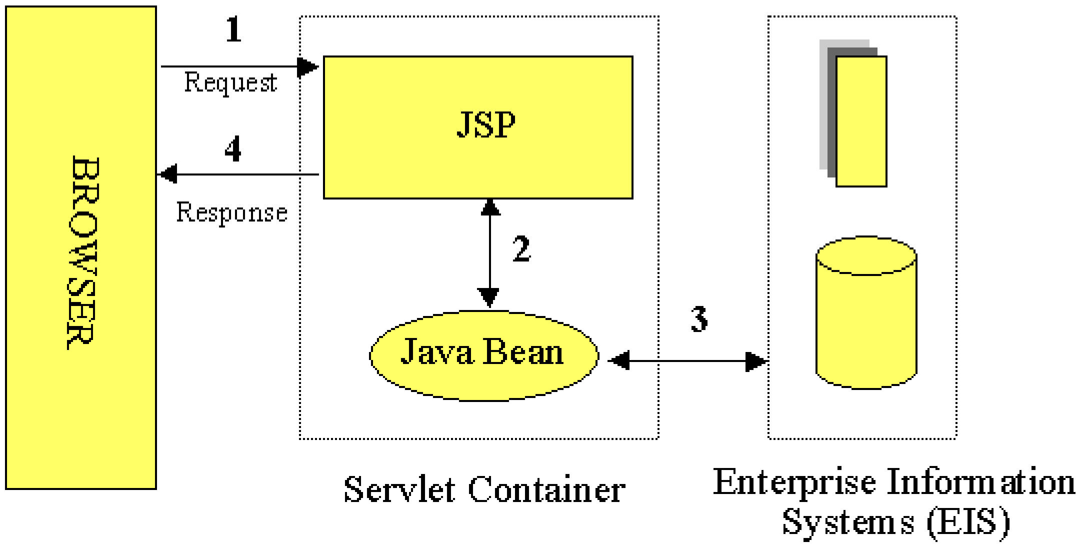


Figure 10: Model 1 Architecture

In the Model 1 architecture, the incoming request from a web browser is sent directly to the JSP page, which is responsible for processing it and replying back to the client. There is still separation of presentation from content, because all data access is performed using beans.

Although the Model 1 architecture is suitable for simple applications, it may not be desirable for complex implementations. Indiscriminate usage of this architecture usually leads to a significant amount of scriptlets or Java code embedded within the JSP page, especially if there is a significant amount of request processing to be performed. While this may not seem to be much of a problem for Java developers, it is certainly an issue if your JSP pages are created and maintained by designers--which is usually the norm on large projects. Another downside of this architecture is that each of the JSP pages must be individually responsible for managing application state and verifying authentication and security.

Model 2 architecture

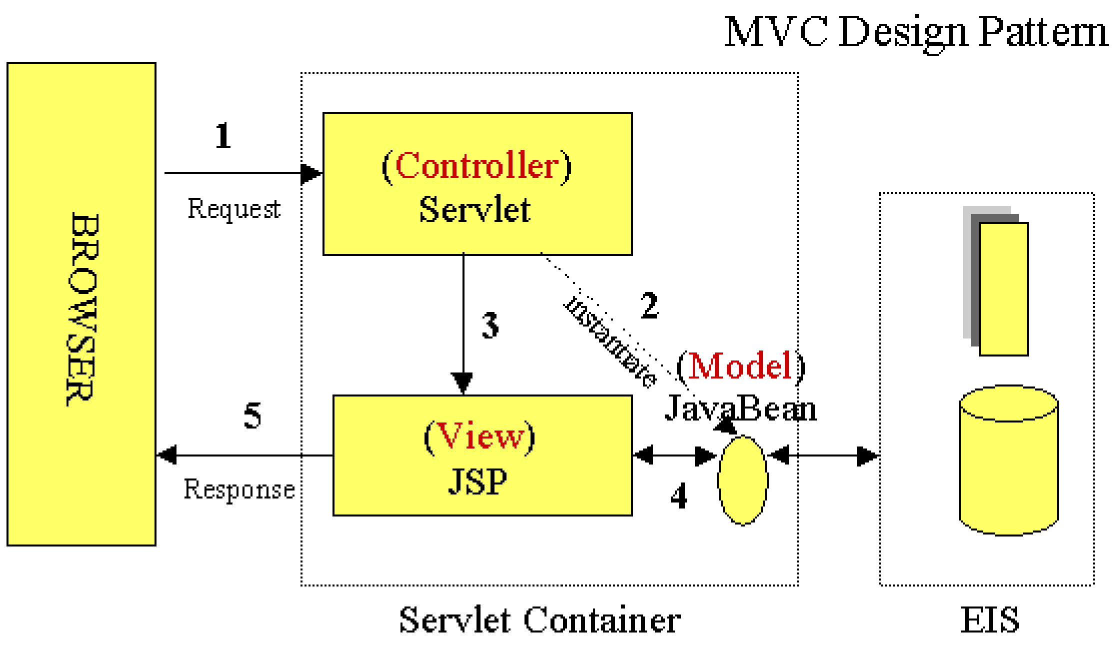


Figure 11: Model 2 Architecture

The Model 2 architecture, shown above, is a server-side implementation of the popular Model/View/Controller design pattern. Here, the processing is divided between presentation and front components. Presentation components are JSP pages that generate the HTML/XML response that determines the user interface when rendered by the browser. Front components (also known as controllers) do not handle any presentation issues, but rather, process all the HTTP requests. Here, they are responsible for creating any beans or objects used by the presentation components, as well as deciding, depending on the user's actions, which presentation component to forward the request to. Front components can be implemented as either a servlet or JSP page.

The advantage of this architecture is that there is no processing logic within the presentation component itself; it is simply responsible for retrieving any objects or beans that may have been previously created by the controller, and extracting the dynamic content within for insertion within its static templates. Consequently, this clean separation of presentation from content leads to a clear delineation of the roles and responsibilities of the developers and page designers on the programming team. Another benefit of this approach is that the front components present a single point of entry into the application, thus making the management of application state, security, and presentation uniform and easier to maintain.

JSP Implicit Objects

JSP implicit objects are created during the translation phase of JSP to the servlet. These objects can be directly used in scriplets that goes in the service method. They are created by the container automatically, and they can be accessed using objects. There are 9 types of implicit objects available in the container.

Following table lists out the nine Implicit Objects that JSP supports −

S.No.	Object & Description
1	**request** This is the **HttpServletRequest** object associated with the request.
2	**response** This is the **HttpServletResponse** object associated with the response to the client.
3	**out** This is the **PrintWriter** object used to send output to the client.
4	**session** This is the **HttpSession** object associated with the request.
5	**application** This is the **ServletContext** object associated with the application context.
6	**config** This is the **ServletConfig** object associated with the page.
7	**pageContext** This encapsulates use of server-specific features like higher performance **JspWriters**.
8	**page**

	This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class.
9	**Exception** The **Exception** object allows the exception data to be accessed by designated JSP.

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();
```

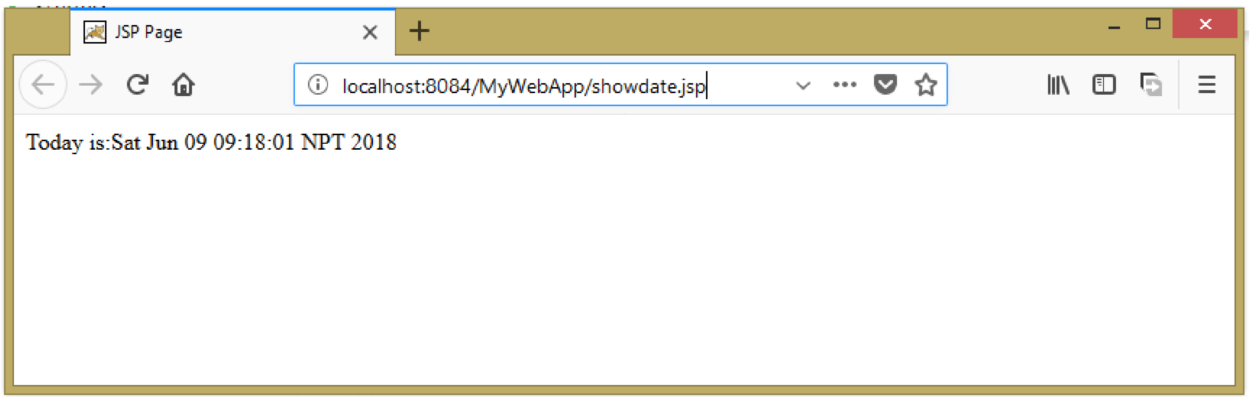
But in JSP, you don't need to write this code.

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>
```



```
Today is:Sat Jun 09 09:18:01 NPT 2018
```

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.

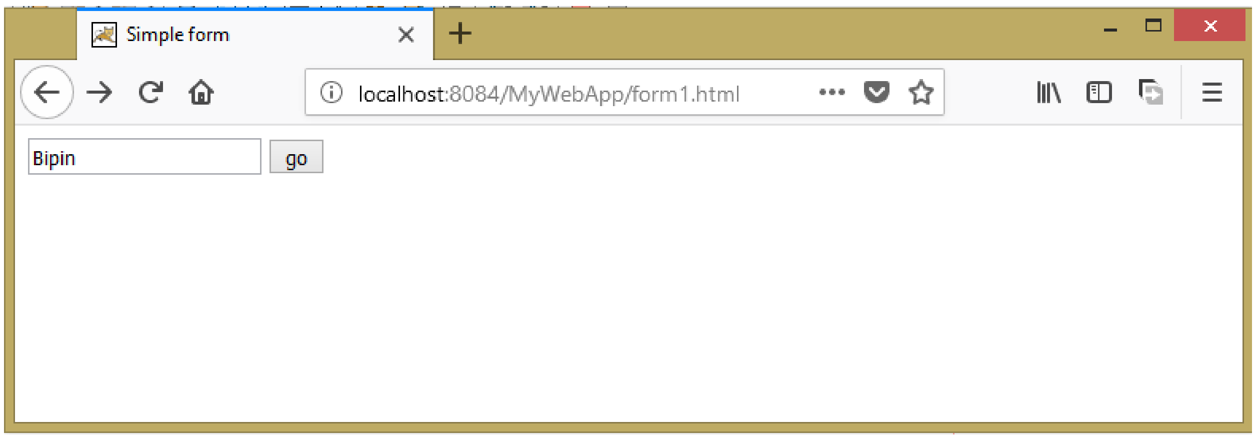
Let's see the simple example of request implicit object where we are printing the name of the user with welcome message.

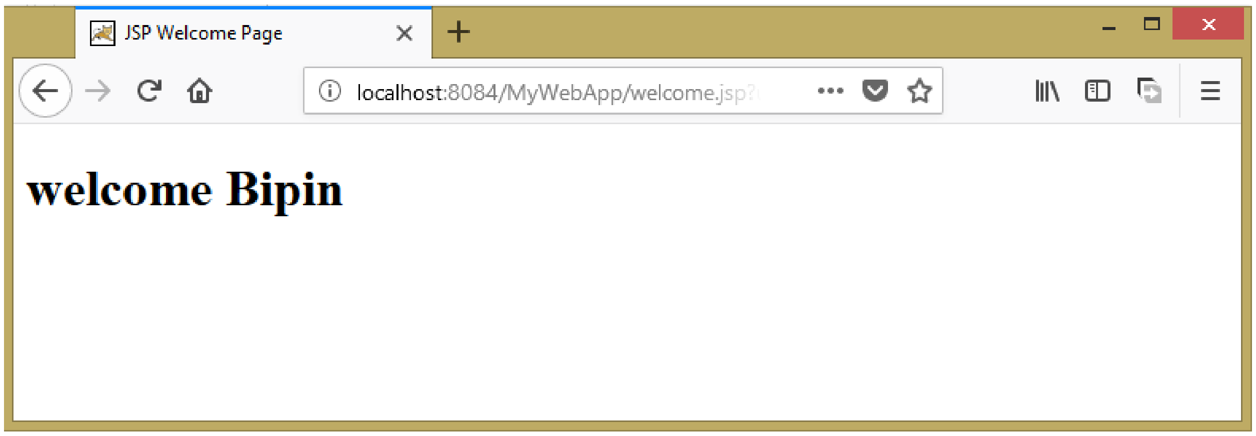
form1.html

```
<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>
```





welcome Bipin

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.

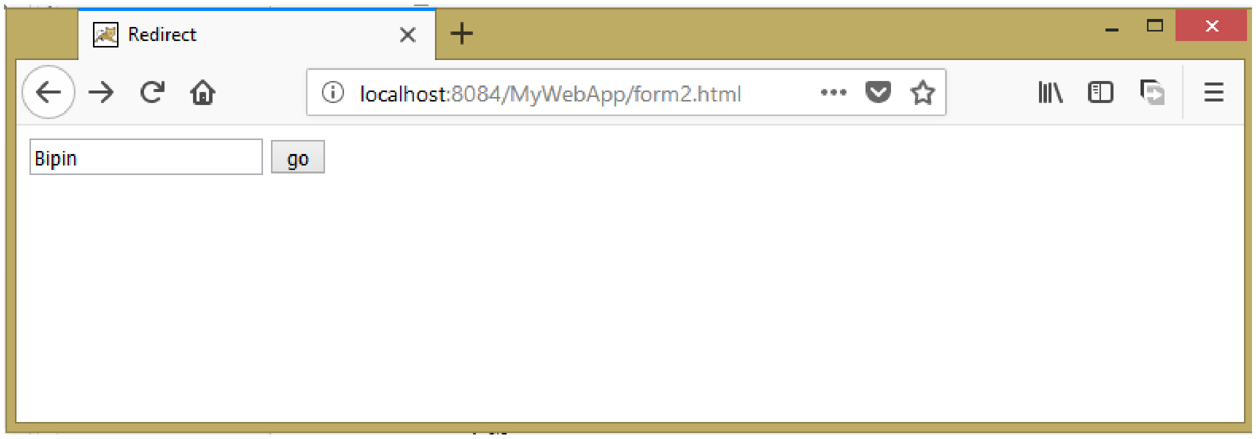
Let's see the example of response implicit object where we are redirecting the response to the 'wordlover'.

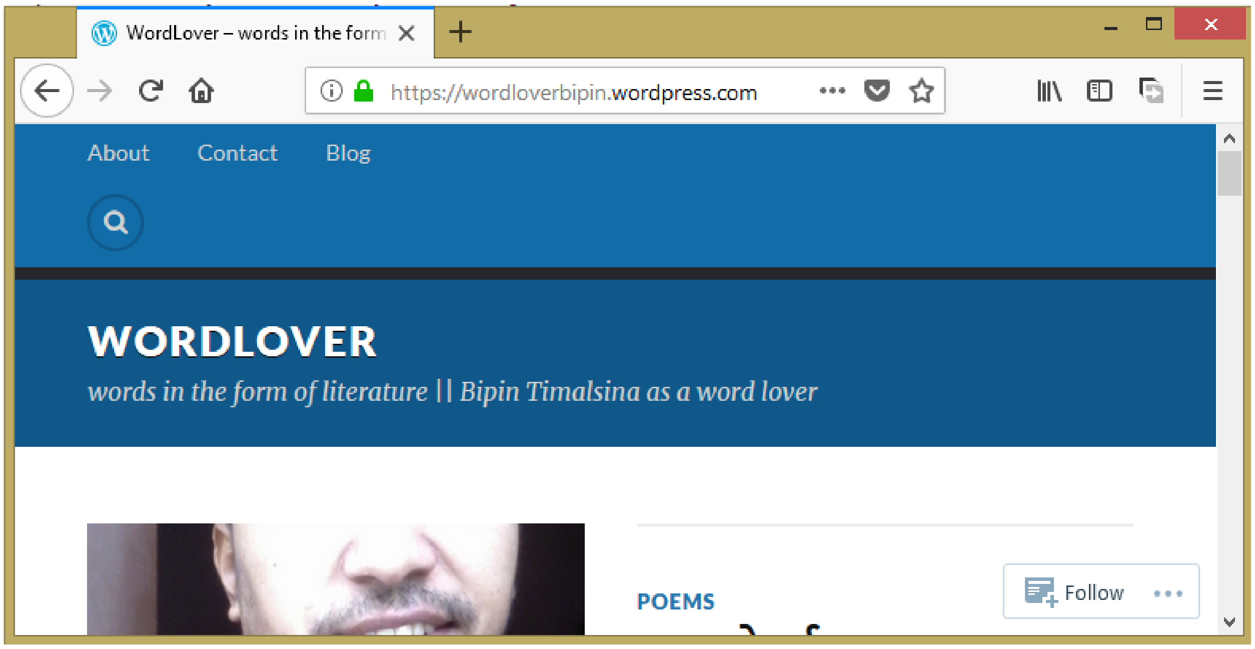
form2.html

```
<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>
```




WordLover – words in the form
https://wordloverbipin.wordpress.com
About
Contact
Blog
WORDLOVER
words in the form of literature || Bipin Timalsina as a word lover
POEMS
Follow

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 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.

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

<% (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 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.

Scripting

JavaServer Pages often present dynamically generated content as part of an XHTML document that is sent to the client in response to a request. In some cases, the content is static but is output only if certain conditions are met during a request (e.g., providing values in a form that submits a request). **JSP programmers can insert Java code and logic in a JSP using scripting.**

Scripting Components

The JSP scripting components include **scriptlets, comments, expressions, declarations and escape sequences.**

Scriptlets are blocks of code delimited by **<% and %>.** They contain Java statements that the container places in method _jspService at translation time.

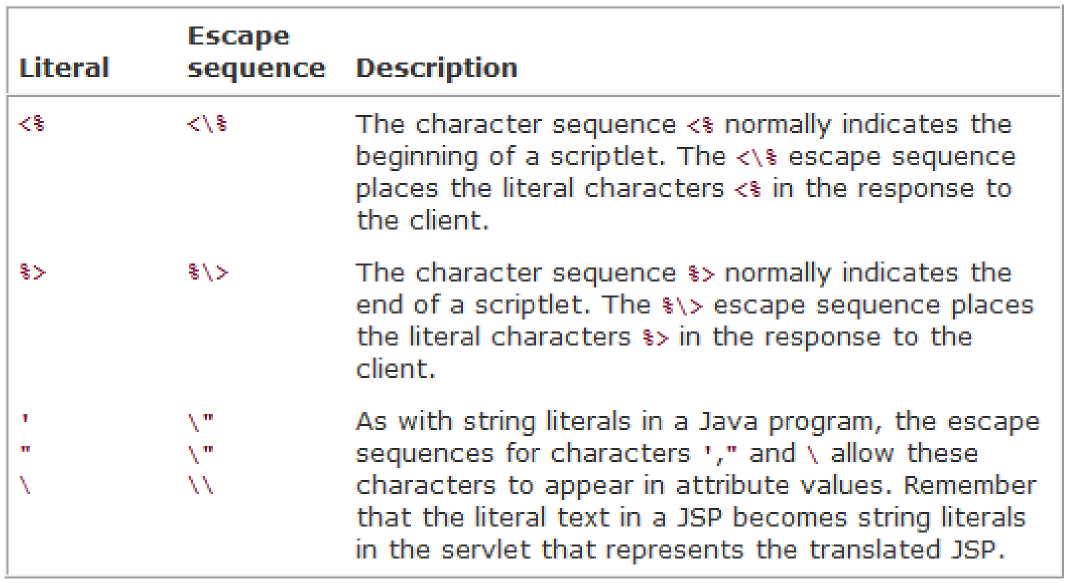
JSPs support three **comment** styles: **JSP comments, XHTML comments and scripting-language comments. JSP comments** are delimited by **<%-- and --%>.** These can be placed throughout a JSP, but not inside scriptlets. **XHTML comments** are delimited with **<!-- and -->.** These, too, can be placed throughout a JSP, but not inside scriptlets. **Scripting language comments** are currently **Java comments**, because Java currently is the only JSP scripting language. Scriptlets can use Java's **end-of-line //** comments and traditional comments (delimited by **/* and */).** JSP comments and scripting-language comments are ignored and do not appear in the response to a client. When clients view the source code of a JSP response, they will see only the XHTML comments in the source code. The different comment styles are useful for separating comments that the user should be able to see from those that document logic processed on the server.

JSP expressions are delimited by **<%= and %>** and contain a Java expression that is evaluated when a client requests the JSP containing the expression. The container converts the result of a JSP expression to a String object, then outputs the String as part of the response to the client.

Declarations, delimited by **<%! and %>,** enable a JSP programmer to define variables and methods for use in a JSP. Variables become instance variables of the servlet class that represents the translated JSP. Similarly, methods become members of the class that represents the translated JSP. Declarations of variables and methods in a JSP use Java syntax. Thus, a variable declaration must end with a semicolon, as in

```
<%! int counter = 0; %>
```

Special characters or character sequences that the JSP container normally uses to delimit JSP code can be included in a JSP as literal characters in scripting elements, fixed template data and attribute values using **escape sequences**. Figure below shows the literal character or characters and the corresponding escape sequences and discusses where to use the escape sequences.



Literal	Escape sequence	Description
<%	<\%	The character sequence <% normally indicates the beginning of a scriptlet. The <\% escape sequence places the literal characters <% in the response to the client.
%>	%\>	The character sequence %> normally indicates the end of a scriptlet. The %\> escape sequence places the literal characters %> in the response to the client.
' " \	\" \" \\	As with string literals in a Java program, the escape sequences for characters '," and \ allow these characters to appear in attribute values. Remember that the literal text in a JSP becomes string literals in the servlet that represents the translated JSP.

fig. JSP escape sequences

- **JSP Expressions:** It is a small java code which you can include into a JSP page. The syntax is "<%= some java code %>"
- **JSP Scriptlet:** The syntax for a scriptlet is "<% some java code %>". You can add 1 to many lines of Java code in here.
- **JSP Declaration**: The syntax for declaration is "<%! Variable or method declaration %>", in here you can declare a variable or a method for use later in the code.

JSP Expressions

Using the JSP Expression you can compute a small expression, always a single line, and get the result included in the HTML which is returned to the browser. Using the code we have previously written, let's explore expressions.

Code

The time on the server is <%= new java.util.Date() %>

- You can also make use of mathematical expressions in JSP.

  The Expression: 25 multiplied to 4: <%= 25*4 %>

  The HTML: 25 multiplied to 4: 100

- You can also have Boolean expressions in JSP.

The Expression: Is 75 less than 69? <%= 75 <69 %>

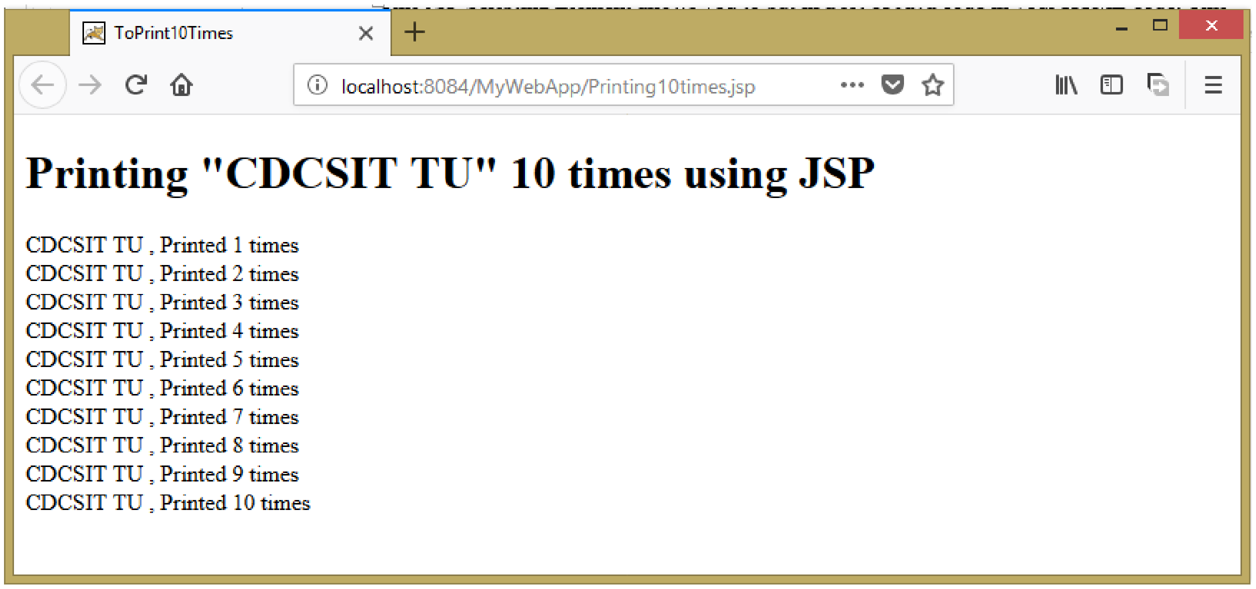
The HTML: Is 75 less than 69? False

JSP Scriptlets

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. The syntax is pretty much the same only you don't have to put in an equal sign after the opening % sign.

Example:

```
<body>
    <h1>Printing "CDCSIT TU" 10 times using JSP</h1>
    <%
  for(int i=1; i<= 10; i++)
  {
  out.println("CDCSIT TU , Printed "+i+" times"+"<br>");
  }
  %>
  </body>
```



```
Printing "CDCSIT TU" 10 times using JSP

CDCSIT TU , Printed 1 times
CDCSIT TU , Printed 2 times
CDCSIT TU , Printed 3 times
CDCSIT TU , Printed 4 times
CDCSIT TU , Printed 5 times
CDCSIT TU , Printed 6 times
CDCSIT TU , Printed 7 times
CDCSIT TU , Printed 8 times
CDCSIT TU , Printed 9 times
CDCSIT TU , Printed 10 times
```

JSP Declarations

The declarations come in handy when you have a code snippet that you want executed more than once. Using the declaration, you can declare the method in the beginning of the code and then call the same method whenever you need in the same page. The syntax is simple:

```
<%!

//declare a variable or a method

%>
```

Example:

```
<%!

String makeItLower(String data)

{

returndata.toLowerCase();

}

%>
```

Here is how you call it

Lower case “Hello World”:<%= makeItLower(“Hello World”) %>

Output

Lower case “Hello World”: hello world

In the method declaration you have your standard java method with the return type of a String. You take a string as an argument and return it converted to lower case. Later we call this function through a JSP Expression.

JSP directives provide instructions to the JSP engine on how to handle JSP.

A JSP directive affects the structure of the servlet class. It usually has the following form:

```
<%@ directive attribute="value" %>
```

Directives

Directives can have a list of attributes defined in key-value pairs and separated by commas.

Directive Tags

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 another file.
<%@ taglib ... %>	Declares a tag library containing custom actions which can be used in the page.

Standard Tag(Action Element)

JSP specification provides Standard(Action) tags for use within your JSP pages. These tags are used to remove or eliminate scriptlet code from your JSP page because scriplet code are technically not recommended nowadays. It's considered to be bad practice to put java code directly inside your JSP page.

Standard tags begin with the jsp: prefix. There are many JSP Standard Action tag which are used to perform some specific task.

The following are some JSP Standard Action Tags available:

Action Tag	Description
`jsp:forward`	forward the request to a new page Usage : `<jsp:forward page="Relative URL" />`
`jsp:useBean`	instantiates a JavaBean Usage : `<jsp:useBean id="beanId" />`

`jsp:getProperty`	retrieves a property from a JavaBean instance.
	Usage :
	`<jsp:useBean id="beanId" ... />`
	`...`
	`<jsp:getProperty name="beanId" property="someProperty" .../>`
	Where, **beanName** is the name of pre-defined bean whose property we want to access.

`jsp:setProperty`	store data in property of any JavaBeans instance.
	Usage :
	`<jsp:useBean id="beanId" ... />`
	`...`
	`<jsp:setProperty name="beanId" property="someProperty" value="some value"/>`
	Where, **beanName** is the name of pre-defined bean whose property we want to access.

`jsp:include`	includes the runtime response of a JSP page into the current page.
`jsp:plugin`	Generates client browser-specific construct that makes an OBJECT or EMBED tag for the Java Applets
`jsp:fallback`	Supplies alternate text if java plugin is unavailable on the client. You can print a message using this, if the included jsp plugin is not loaded.
`jsp:element`	Defines XML elements dynamically
`jsp:attribute`	defines dynamically defined XML element's attribute
`jsp:body`	Used within standard or custom tags to supply the tag body.
`jsp:param`	Adds parameters to the request object.
`jsp:text`	Used to write template text in JSP pages and documents.

	Usage : `<jsp:text>Template data</jsp:text>`

JSP comments

JSP comment elements are used to hide your code from the "view page source". However, you can use HTML comment tags in JSP page but, when user of your website will choose the "view page source" then they can see your commented code. The JSP comment is denoted by the special character <%--   --%>. The special character '<%--' specifies the opening comment element and the special character '--%>' specifies the end of the comment tag.

HTML comment tag is denoted by the special characters '<!-- -->' where, '<!--' specifies the start tag and '-->' specifies the end tag.

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.

Exception Handling is a process of handling exceptional condition that might occur in your application. Exception Handling in JSP is much easier than Java Technology exception handling. Although JSP Technology also uses the same exception class objects.

It is quite obvious that you don’t want to show error stack trace to any random user surfing your website. You can't prevent all errors in your application but you can at least give a user friendly error response page.

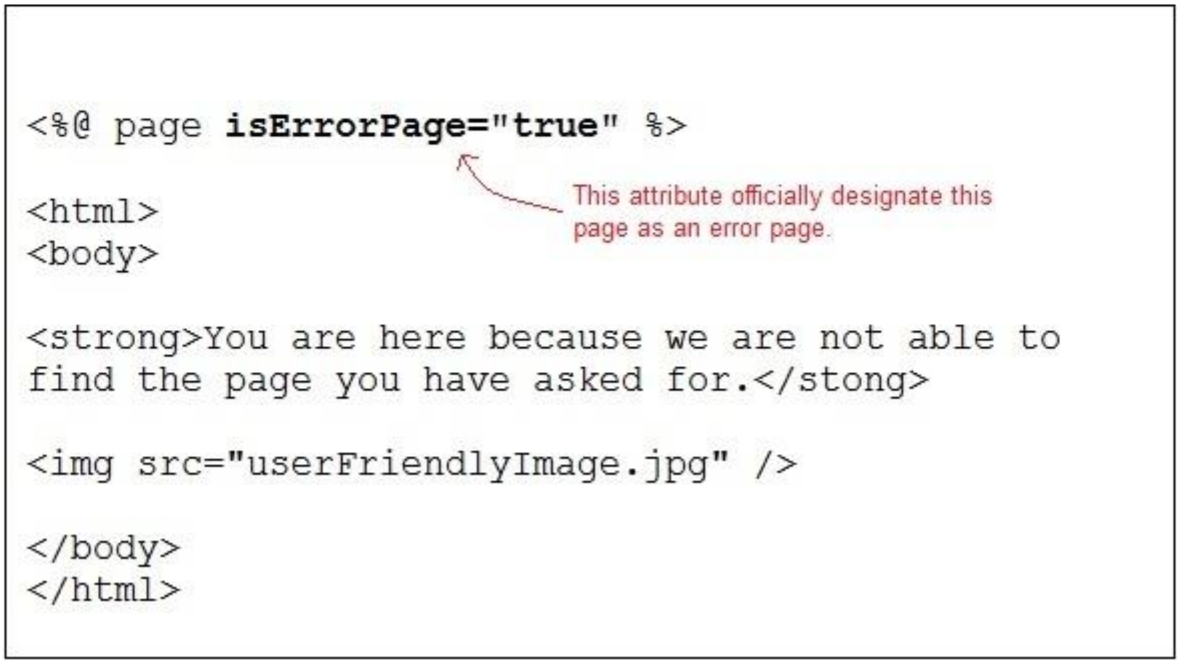
JSP provide 3 different ways to perform exception handling:

- Using **isErrorPage** and **errorPage** attribute of page directive.
- Using **<error-page>** tag in **Deployment Descriptor**.
- Using simple `try...catch` block.

Example of isErrorPage and errorPage attribute

`isErrorPage` attribute in page directive officially appoints a JSP page as an error page.

error.jsp

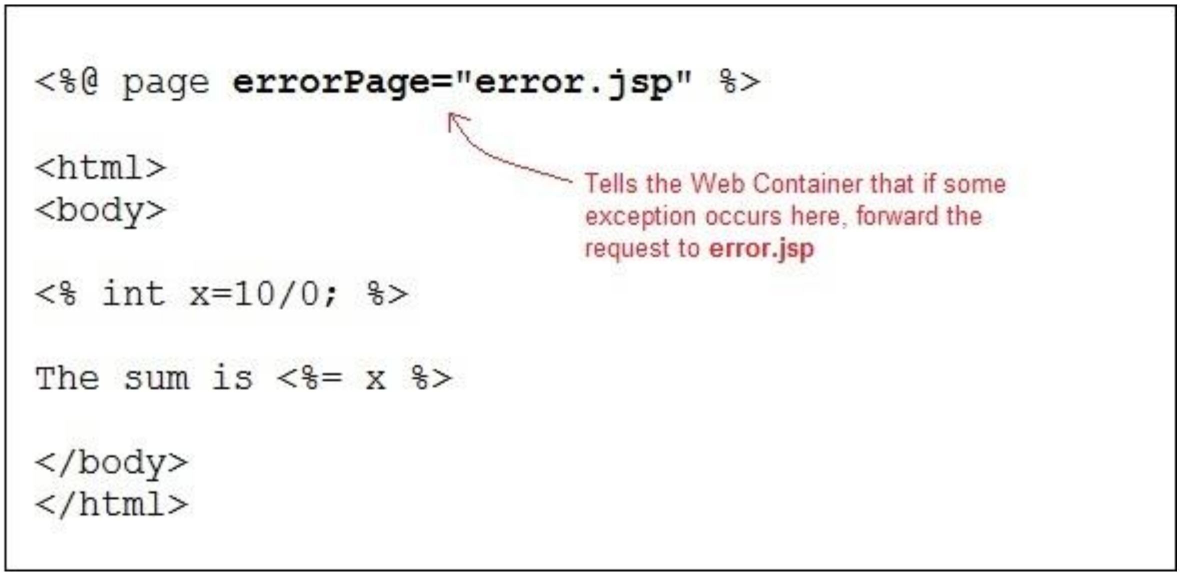


```
<%@ page isErrorPage="true" %>

<html>
<body>

<strong>You are here because we are not able to
find the page you have asked for.</stong>

<img src="userFriendlyImage.jpg" />

</body>
</html>
```

This attribute officially designate this page as an error page.

`errorPage` attribute in a page directive informs the Web Container that if an exception occurs in the current page, forward the request to the specified error page.

sum.jsp



```
<%@ page errorPage="error.jsp" %>

<html>
<body>

<% int x=10/0; %>

The sum is <%= x %>

</body>
</html>
```

Tells the Web Container that if some exception occurs here, forward the request to **error.jsp**

Whenever an exception occurs in sum.jsp page the user is redirected to the error.jsp page, where either you can display a nice message, or you can also print the exception trace into a file/database in the background, to check later what caused the error.

Declaring error page in Deployment Descriptor

You can also declare error pages in the DD for the entire Web Apllication. Using <error-page> tag in the Deployment Descriptor. You can even configure different error pages for different exception types, or HTTP error code type(503, 500 etc).

Declaring an error page for all type of exception

```
<error-page>
<exception-type>java.lang.Throwable</exception-type>
<location>/error.jsp</location>
</error-page>
```

Declaring an error page for more detailed exception

```
<error-page>
<exception-type>java.lang.ArithmeticException</exception-type>
<location>/error.jsp</location>
</error-page>
```

Declaring an error page based on HTTP Status code

```
<error-page>

<error-code>404</error-code>

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

Using the try...catch block

Using `try...catch` block is just like how it is used in Core Java.

```
<html>

<head>

  <title>Try...Catch Example</title>

</head>

<body>

 <%

  try{

    int i = 100;

    i = i / 0;

    out.println("The answer is " + i);

  }

  catch (Exception e){

    out.println("An exception occurred: " + e.getMessage());

  }

 %>

</body>

</html>
```

JSP Session Management

In this section, we will discuss session tracking in JSP. HTTP is a "stateless" protocol which means each time a client retrieves a Webpage, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.

Maintaining Session Between Web Client And Server

Let us now discuss a few options to maintain the session between the Web Client and the Web Server −

Cookies

A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.

This may not be an effective way as the browser at times does not support a cookie. It is not recommended to use this procedure to maintain the sessions.

Hidden Form Fields

A web server can send a hidden HTML form field along with a unique session ID as follows −

```
<input type = "hidden" name = "sessionid" value = "12345">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or the POST data. Each time the web browser sends the request back, the session_id value can be used to keep the track of different web browsers.

This can be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

URL Rewriting

You can append some extra data at the end of each URL. This data identifies the session; the server can associate that session identifier with the data it has stored about that session.

For example, with http://tutorialspoint.com/file.htm;sessionid=12345, the session identifier is attached as sessionid = 12345 which can be accessed at the web server to identify the client.

URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies. The drawback here is that you will have to generate every URL dynamically to assign a session ID though page is a simple static HTML page.

The session Object

Apart from the above mentioned options, JSP makes use of the servlet provided HttpSession Interface. This interface provides a way to identify a user across.

- a one page request or

- visit to a website or
- store information about that user

By default, JSPs have session tracking enabled and a new HttpSession object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows −

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

The JSP engine exposes the HttpSession object to the JSP author through the implicit session object. Since session object is already provided to the JSP programmer, the programmer can immediately begin storing and retrieving data from the object without any initialization or getSession().

Example

form3.jsp

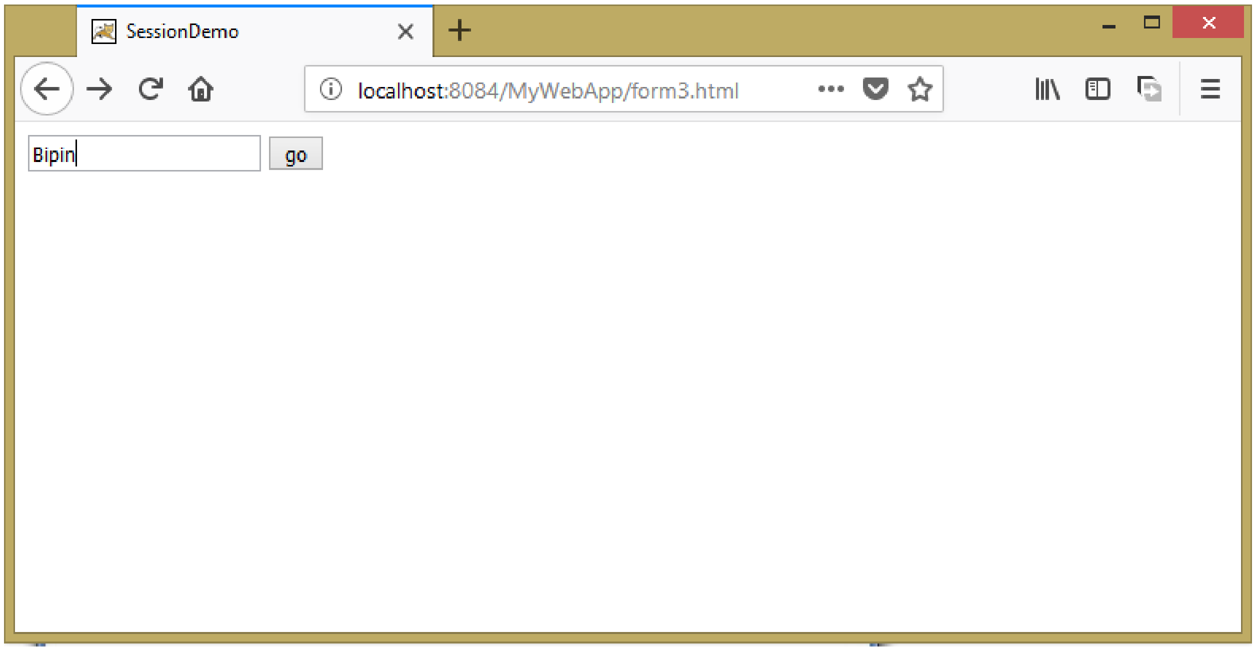
```
<body>
  <form action="greeting.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
   </body>
```

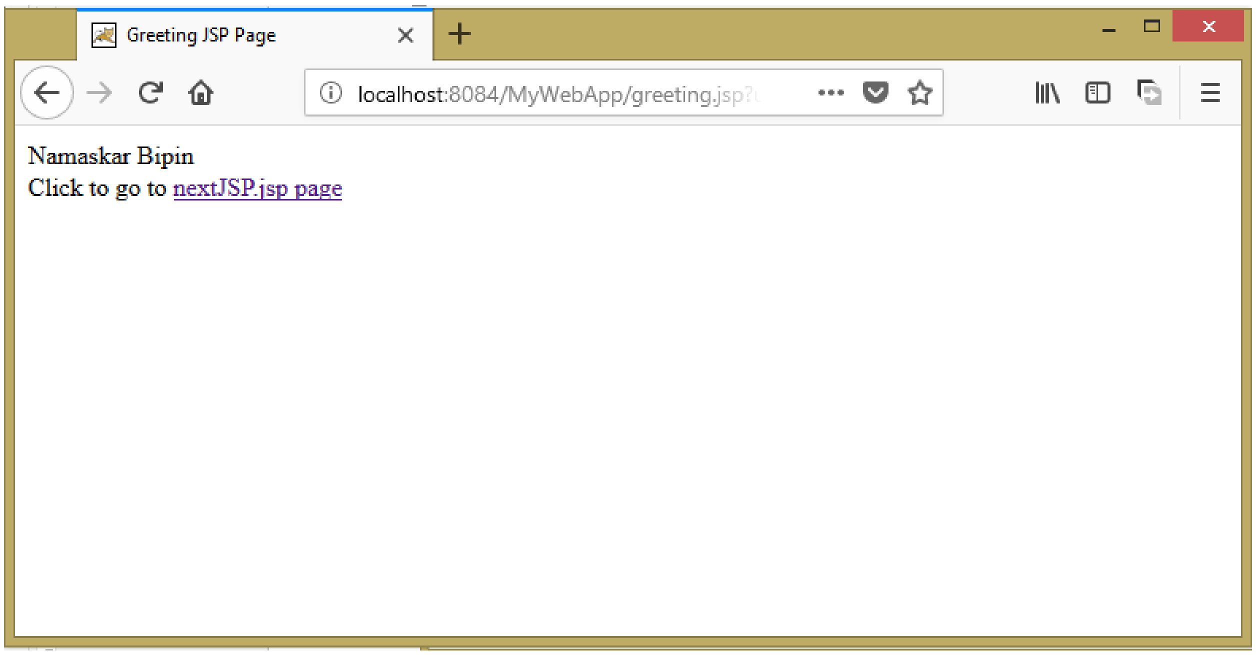
greeting.jsp

```
<body>
              <%
        String name=request.getParameter("uname");
        out.print("Namaskar "+name);
        session.setAttribute("user",name);  %>
                <br>Click to go to  <a href="nextJSP.jsp">nextJSP.jsp page</a>
   </body>
```

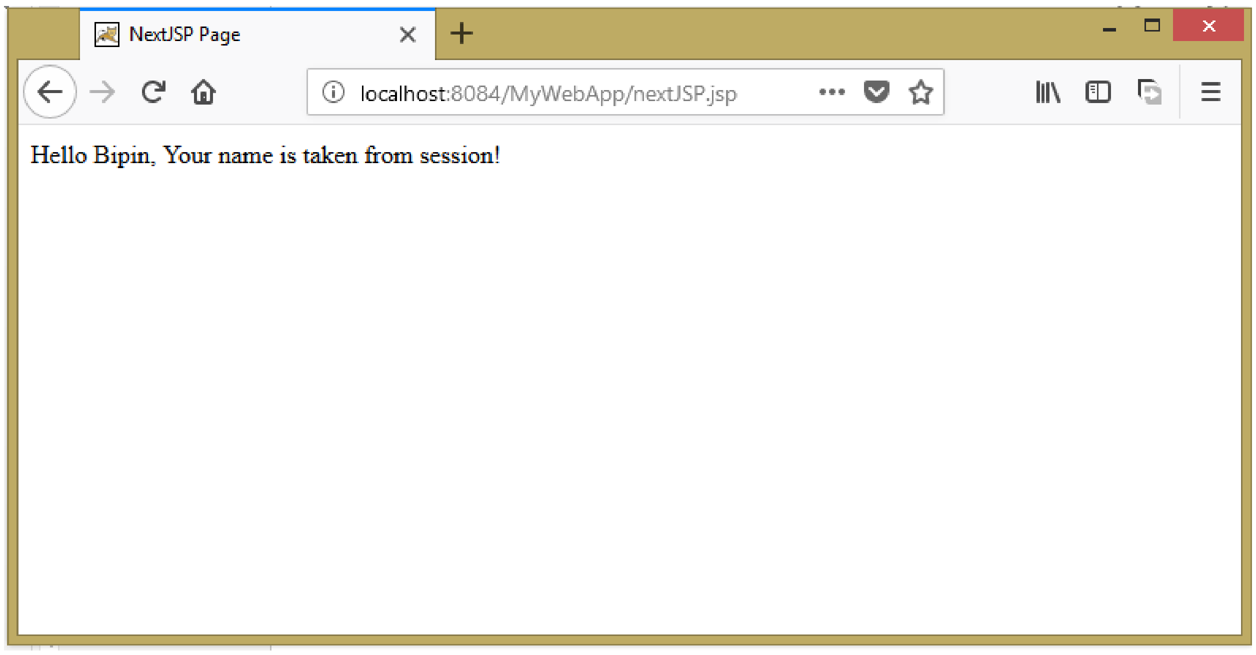
nextJSP.jsp

```
<body>
        <%
 String name=(String)session.getAttribute("user");
out.print("Hello "+name+", Your name is taken from session!");
 %>
    </body>
```





Namaskar Bipin
Click to go to nextJSP.jsp page



Hello Bipin, Your name is taken from session!

Synchronization Issue

By default JSP is not thread safe since:

- ✓ multiple threads are allowed to execute single instance of a JSP at the same time
- ✓ two or more threads may access (read/modify) the same instance variable of the JSP instance at the sametime (which may corrupt the value of the instance variable)

Solution

- ✓ Avoid JSP declaration (<%! declaration %> tag) that creates instance variables; instead use local variables (<% declaration %>) – note the absence of exclamation sign in the latter
- ✓ Use <%@ page isThreadSafe=”false” %> to implement SingleThreadModel interface in your JSP (ServletContext and HttpSession objects are still not thread safe)
- ✓ Use synchronized block or synchronized method to access shared data (instance variables, ServletContext and objects, HttpSession objects) – use code example from Multithreading chapter in JSP page

Creating and Processing Forms

To send information to web server, we can use two methods: **GET Method and POST Method**.

The GET method sends the encoded user information separated by the ? character appended to the page URL.

```
http://www.java2s.com/hello?key1=value1&key2=value2
```

The **GET** method is the default method to send information to web server. Since the GET method appends plain text string to the URL. We should avoid using GET method to send password or other sensitive information to the server.

The GET method also has size limitation. We can send only 1024 characters in a request string. This information sent is accessible **getQueryString()** and **getParameter()** method**s** of request object.

POST method is more reliable method of sending information to the server. This method sends the information as a separate message.

JSP handles this type of requests using **getParameter()** method to read simple parameters and **getInputStream()** method to read binary data stream coming from the client.

JSP handles form data using the following methods.

Method	**Description**
getParameter()	returns the value of a form parameter.
getParameterValues()	returns multiple values, for example checkbox.
getParameterNames()	returns a complete list of all parameters.
getInputStream()	read binary data stream coming from the client.

GET parameters from URL

The following code shows how to get parameters using GET method.

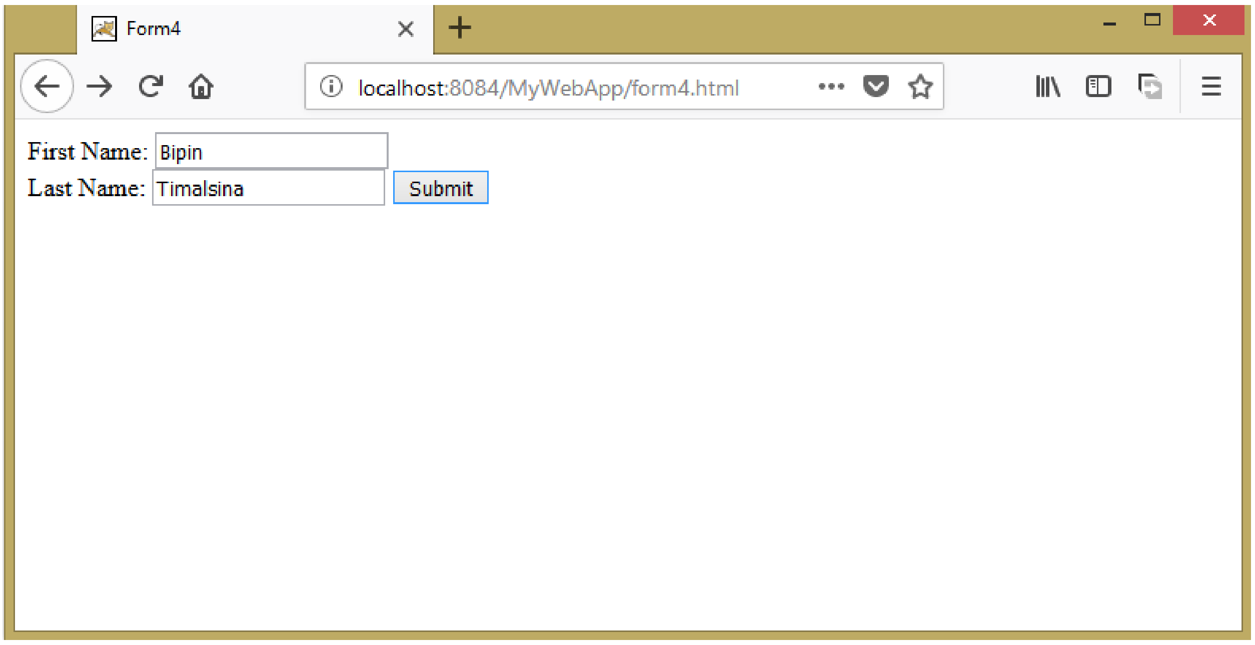
The following code uses the **getParameter()** method to read parameters.

getDemo.jsp

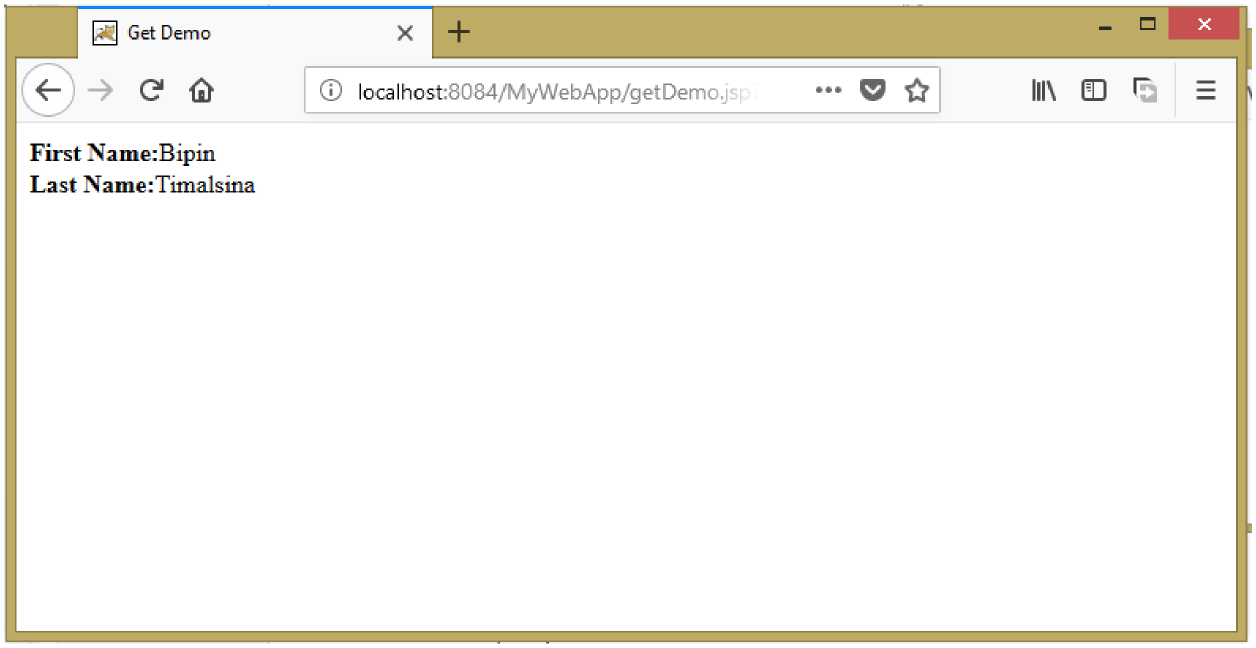
```
<html>

<body>

<b>First Name:</b><%= request.getParameter("first_name")%><br/>

<b>Last  Name:</b><%= request.getParameter("last_name")%>

</body>

</html>
```

form4.jsp

```
<html>

    <body>

    <form action="getDemo.jsp" method="GET">

         First Name: <input type="text" name="first_name">

    <br />

         Last Name: <input type="text" name="last_name" />

    <input type="submit" value="Submit" />

    </form>

    </body>

</html>
```



First Name: Bipin
Last Name: Timalsina



First Name: Bipin
Last Name: Timalsina

POST Method Example with Form

Create a html page as follows and use POST methods for the form. Save it as form5.html.

```
<html>
<body>
<form action="postDemo.jsp" method="POST">
First Name: <input type="text" name="first_name">
```

```
<br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

postDemo.jsp

```
<html>
<body>
<b>First Name:</b><%= request.getParameter("first_name")%><br/>
<b>Last  Name:</b><%= request.getParameter("last_name")%>
</body>
</html>
```

Passing Checkbox Data to JSP

The following code shows how to use the JSP page to get the checkbox data from a form.

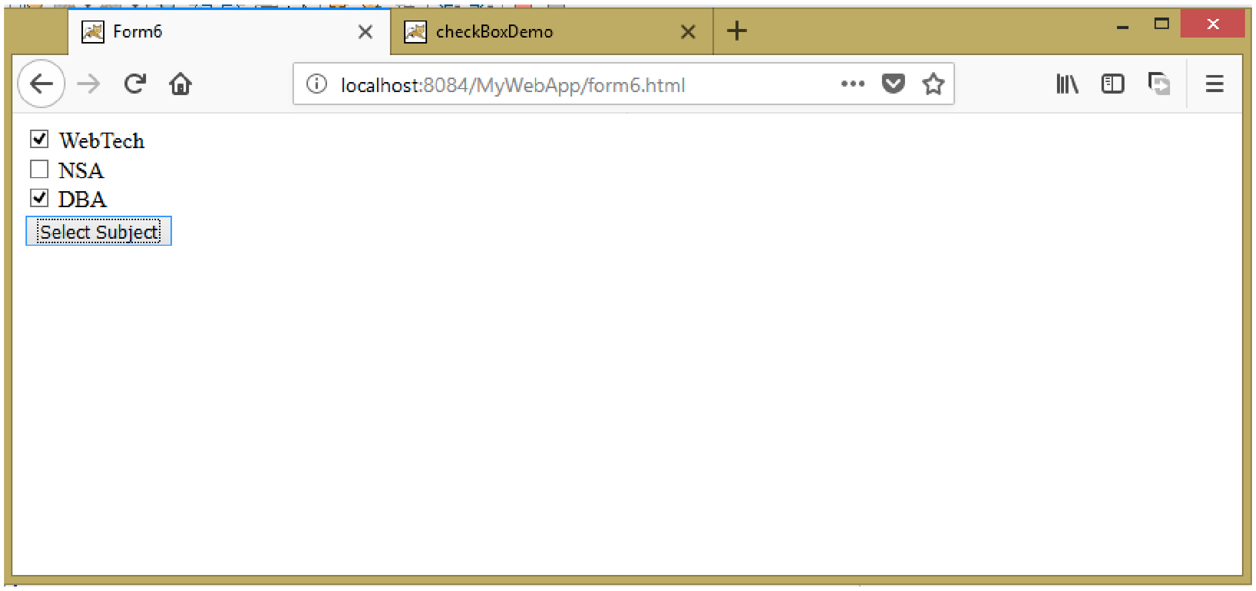
Here is the html form code. It uses the POST method and calls the checkBoxDemo.jsp page

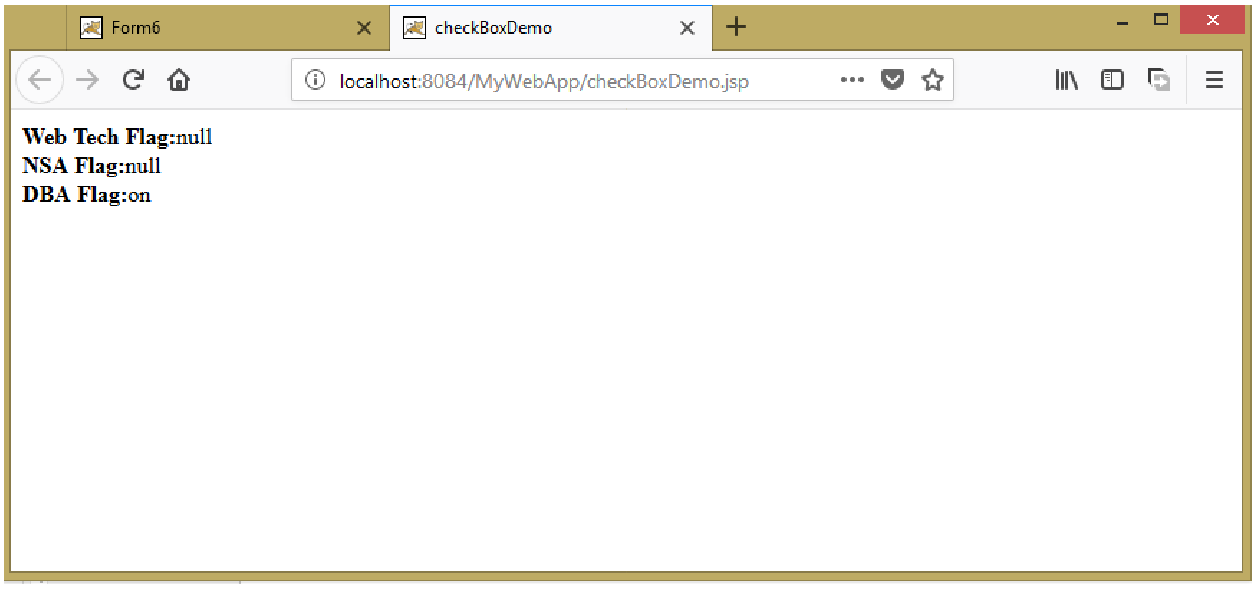
form6.html

```
<html>
<body>
<form action="checkBoxDemo.jsp" method="POST" target="_blank">
    <input type="checkbox" name="webTech" checked="checked" /> WebTech<br>
    <input type="checkbox" name="nsa"/> NSA <br>
    <input type="checkbox" name="dba" /> DBA <br>
<input type="submit" value="Select Subject" />
</form>
</body>
</html>
```

checkBoxDemo.jsp

```
<html>

<body>

<b>Web Tech  Flag:</b><%= request.getParameter("webtech")%><br>

<b>NSA Flag:</b><%= request.getParameter("nsa")%><br>

<b>DBA Flag:</b><%= request.getParameter("dba")%></body>

</html>
```





Web Tech Flag:null
NSA Flag:null
DBA Flag:on

Reading All Form Parameters

Following is a generic example which uses **getParameterNames()** method of **HttpServletRequest** to read all the available form parameters. This method returns an Enumeration that contains the parameter names in an unspecified order.

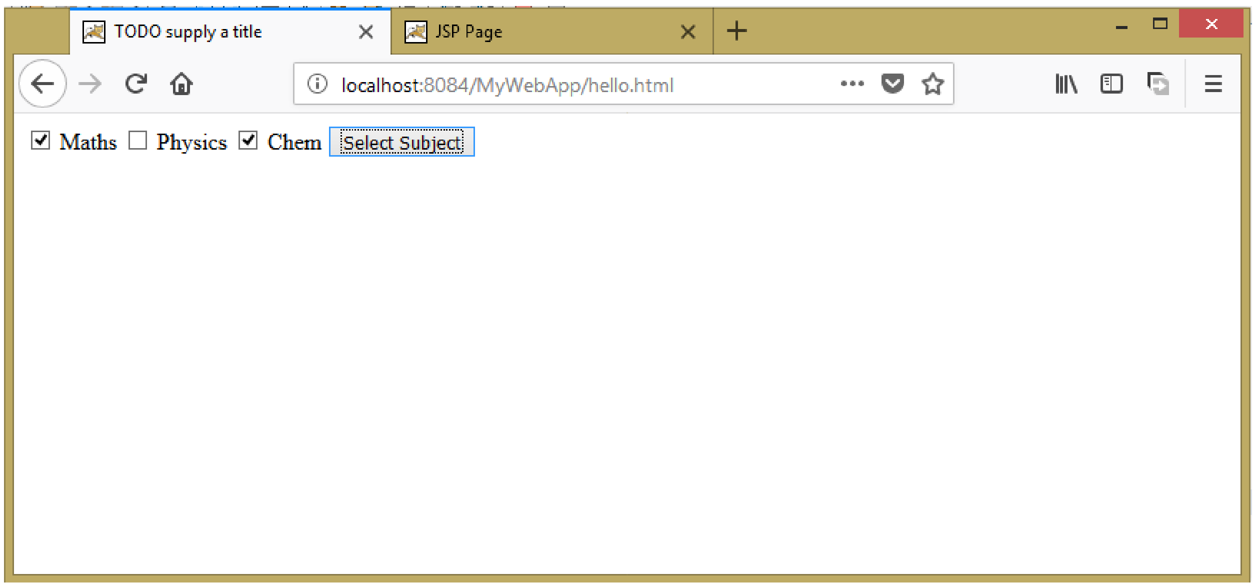
Once we have an Enumeration, we can loop down the Enumeration in the standard manner, using the **hasMoreElements()** method to determine when to stop and using the **nextElement()** method to get each parameter name.

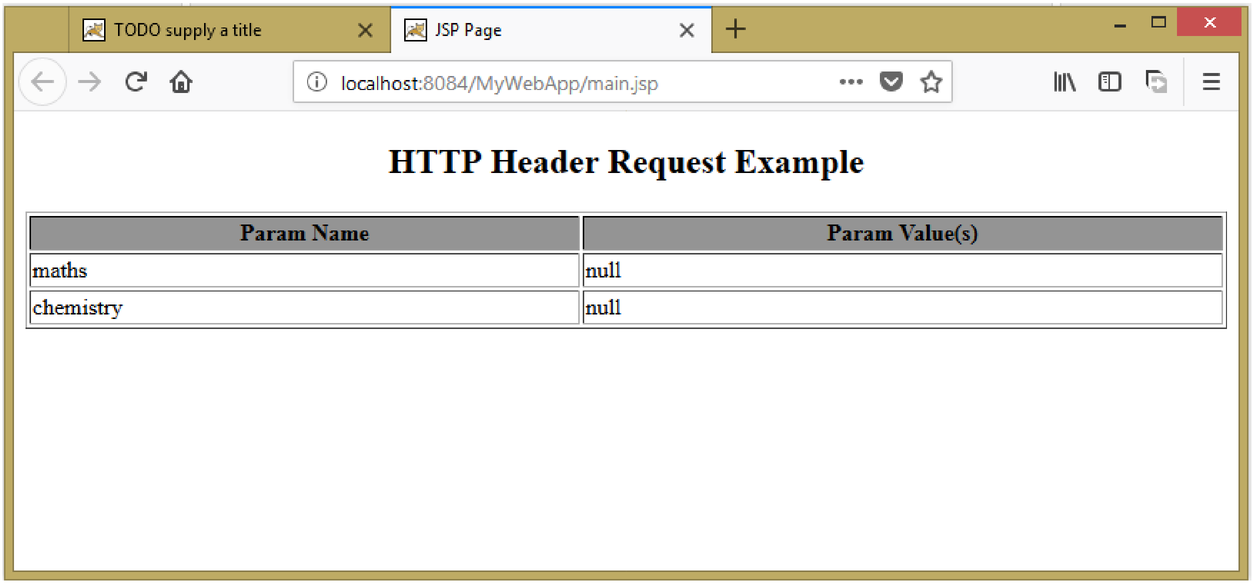
```
<%@ page import = "java.io.*,java.util.*" %>

<html>

   <head>

      <title>HTTP Header Request Example</title>

   </head>

   <body>

      <center>

         <h2>HTTP Header Request Example</h2>

         <table width = "100%" border = "1" align = "center">

            <tr bgcolor = "#949494">

               <th>Param Name</th>

               <th>Param Value(s)</th>

            </tr>

            <%

               Enumeration paramNames = request.getParameterNames();

               while(paramNames.hasMoreElements()) {

                  String paramName = (String)paramNames.nextElement();

                  out.print("<tr><td>" + paramName + "</td>\n");

                  String paramValue = request.getHeader(paramName);

                  out.println("<td> " + paramValue + "</td></tr>\n");

               }

            %>
         </table>
      </center>

   </body>
</html>
```

Following is the content of the **Hello.html –**

```
<html>
   <body>
         <form action = "main.jsp" method = "POST" target = "_blank">
         <input type = "checkbox" name = "maths" checked = "checked" /> Maths
         <input type = "checkbox" name = "physics"  /> Physics
         <input type = "checkbox" name = "chemistry" checked = "checked" /> Chem
         <input type = "submit" value = "Select Subject" />
      </form>
   </body>
</html>
```

Now try calling JSP using the above Hello.htm; this would generate a result something like as below based on the provided input –





HTTP Header Request Example

Param Name	Param Value(s)
maths	null
chemistry	null

You can try the above JSP to read any other form's data which is having other objects like text box, radio button or dropdown, etc.

References

1. javatpoint.com
2. tutorialspoint.com
3. docs.oracle.com
4. javatutorial.net
5. journaldev.com
6. beginnersbook.com
7. oxxus.net
8. way2java.com
9. studytonight.com
10. javapapers.com