

10. Java Server Pages (JSP)/ Sever Technology [Credit: 7 hrs]

10.1. JSP/ Servlet Technology Overview

Applets, Servlets, and Java Server Pages

When we instruct our Web browser to view a page from a Web server on the Internet, our Web browser requests the page from the Web server, the Web server processes the request (which may involve reading the requested page from a file on the hard drive), and then the Web server sends the requested page to our Web browser. Our Web browser *formats*, or *renders*, the *received data to fit on our computer screen*. This interaction is a specific case of the *client/server* model. Our Web browser is the client program, our computer is the *client computer*, the remote website is the *server computer*, and the Web server software running on the remote website is the *server program*.

In the context of a Web application, the *client/server* model is important because Java code can run in two places: on the *client* or on *the server*. There are trade-offs to both approaches.

Server-based programs have easy access to information that resides on the server, such as customer orders or inventory data. Because all of the computation is done on the server and results are transmitted to the client as HTML, a client does not need a powerful computer to run a server-based program.

On the other hand, a *client-based* program may require a more powerful client computer, because all computation is performed locally. However, richer interaction is possible, because the client program has access to local resources, such as the *graphics display* (e.g., perhaps using Swing) or the operating system. Many systems today are constructed using code that runs on both the client and the server to reap the benefit of both approaches.

Web applications built with Java include: *Java applets*, *Java servlets*, and *Java Server Pages (JSP)*.

Java applets run on the *client computer*.

JavaScript, which is a different language than Java despite its similar name, also runs on the client computer as part of the Web browser.

Java servlets and *Java Server Pages* run on the *server*.

Java Server pages which are a dynamic version of *Java servlets*. *Servlets must be compiled before they can run, just like a normal Java program*. In contrast, *JSP code is embedded with the corresponding HTML and is compiled "on the fly" into a servlet when the page is requested*. This flexibility can make it easier to develop Web applications using JSP than with Java servlets.

The following figure 10.1, figure 10.2 and figure 10.3 shows that the running a Java applet, Java Servlet and JSP program.

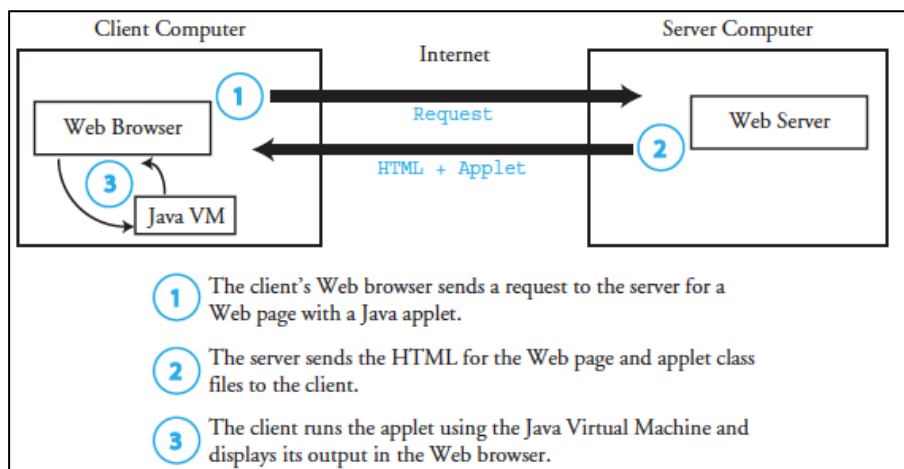


Figure 10.1: Running a Java Applet

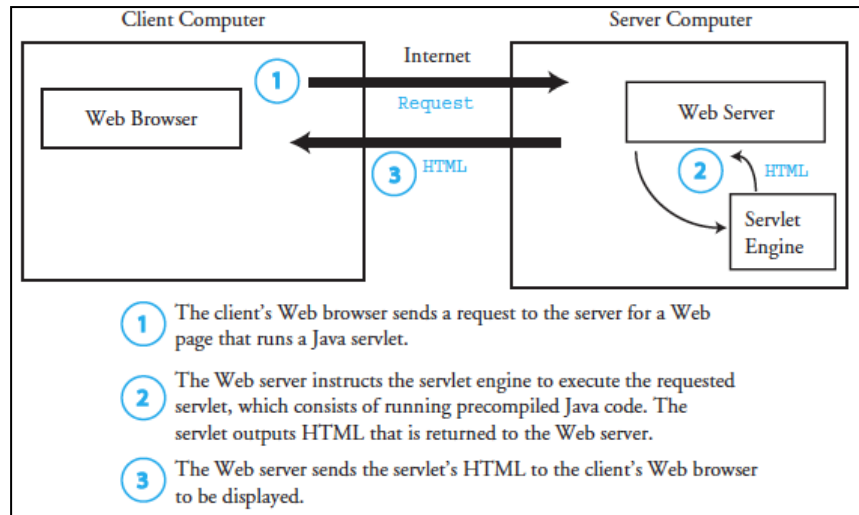


Figure 10.2: Running a Java Servlet

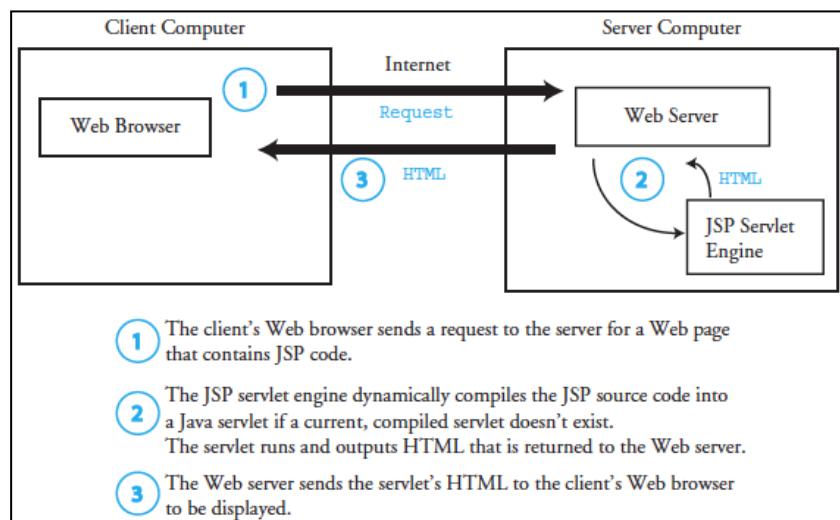


Figure 10.3: Running a Java Server Page (JSP) Program

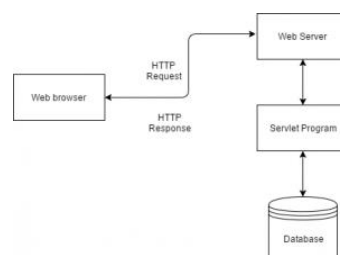
10.2. Servlet Life Cycle, Creating and deploying new Servlet

10.2.1. Servlet

Servlets are the Java programs that runs on the Java-enabled web server or application server. They 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. They are used to handle the request obtained from the web server, process the request, produce the response, then send response back to the web server.

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

The following diagram shows the position of Servlets in a Web Application.



10.2.2. Servlet advantages

Advantages of Servlet over CGI

CGI is actually an external application which is written by using any of the programming languages like C or C++ and this is responsible for processing client requests and generating dynamic content.

Before introduction of Java Servlet API, CGI technology was used to create dynamic web applications. CGI technology has many drawbacks such as creating separate process for each request, platform dependent code (C, C++), high memory usage and slow performance.

Java Servlets often serve the same purpose as programs implemented using the CGI. CGI has several limitations such as performance, scalability, reusability, etc. that a servlet doesn't have.

Servlets offer several advantages in comparison with the CGI.

1. Performance is significantly better. Servlets execute within the address space of a Web server. It is not necessary to create a separate process to handle each client request.
2. Servlets are platform-independent because they are written in Java.
3. Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. So, servlets are trusted.
4. The full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI (Remote Method Invocation) mechanisms that you have seen already.

Difference between Servlet and CGI

Basis for Comparison	Common Gateway Interface	Servlets
1. Basic	Programs are written in the native OS.	Programs employed using Java.
2. Platform dependency	Platform dependent	Does not rely on the platform
3. Creation of process	Each client request creates its own process.	Processes are created depending on the type of the client request.
4. Conversion of the script	Present in the form of executables (native to the server OS).	Compiled to Java Bytecode.
5. Runs on	Separate process	JVM
6. Security	More vulnerable to attacks.	Can resist attacks.
7. Speed	Slower	Faster
8. Processing of script	Direct	Before running the scripts it is translated and compiled.
9. Portability	Cannot be ported	Portable

10.2.3. Servlet Life Cycle

Java Servlet life cycle consists of a series of events that begins when the Servlet container loads Servlet, and ends when the container is closed down. A Servlet container is the part of a web server or an application server that controls a Servlet by managing its life cycle.

A Servlet life cycle can be defined as the entire process from its creation till the destruction. Three methods are central to the life cycle of a servlet: `init()`, `service()` and `destroy()`.

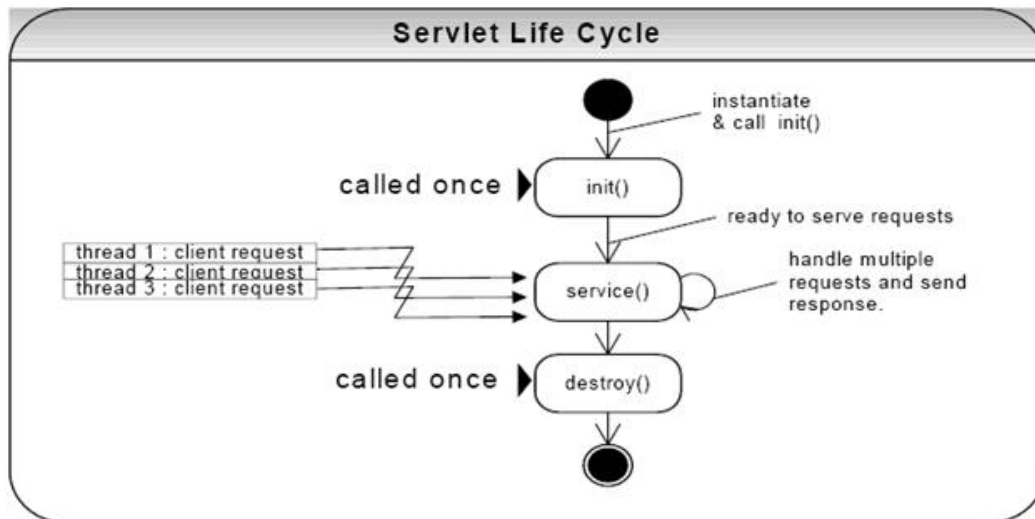


Figure: Servlet Life Cycle

a) Loading and Instantiation

Assume that a user enters a URL to a web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server. The web server receives this HTTP request. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

b) Initialization

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. The `init()` method is called once only.

c) Handling Request

The server invokes the `service()` method of the servlet. This method is called to process the HTTP request (i.e. client's 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.

d) Destroy

The `destroy()` method runs only once during the lifetime of a Servlet. It signals the end of the Servlet instance. 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.

10.2.4 . A simple Servlet

To create a Servlet application, we need to follow the below mentioned steps. These steps are common for all the Web server. In our example we are using Apache Tomcat server. Apache Tomcat is an open source web server for testing servlets and JSP technology.

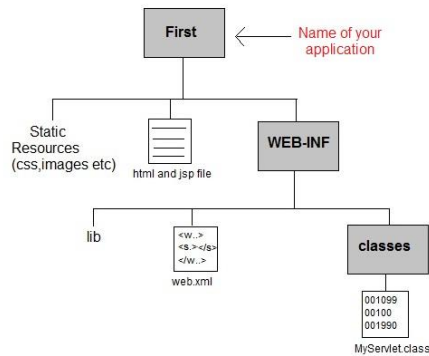
After installing Tomcat Server on your machine follow the below mentioned steps :

1. Create directory structure for your application.
2. Create a Servlet
3. Compile the Servlet
4. Create Deployment Descriptor for your application
5. Start the server and deploy the application

1. Create directory structure for your application.

Create the below directory structure inside **Apache-Tomcat\webapps** directory.

- All HTML, static files (images, css etc) are kept directly under **Web application** folder.
- The **web.xml** (deployment descriptor) file is kept under **WEB-INF** folder.
- While all the Servlet classes are kept inside **classes** folder inside **WEB-INF** folder.
- Here, we make **First** as our application folder.



2. Create a Servlet

Open any text editor (Notepad or Notepad++), write the below code and save it as **HelloServlet.java**

```

import java.io.*;
import javax.servlet.*;
public class HelloServlet extends GenericServlet {
    public void service(ServletRequest request, ServletResponse response) throws
    ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>Hello! This is my First Servlet");
        pw.close();
    }
}
  
```

Let's look closely at this program.

First, note that it imports the **javax.servlet** package. This package contains the classes and interfaces required to build servlets.

Next, the program defines **HelloServlet** as a subclass of **GenericServlet**. The **GenericServlet** class provides functionality that simplifies the *creation of a servlet*. For example, it provides versions of **init()** and **destroy()**, which may be used as is. We need to supply only the **service()** method. Inside **HelloServlet**, the **service()** method (which is inherited from **GenericServlet**) is overridden. This method handles requests from a client.

Notice that the first argument is a **ServletRequest** object. This enables the servlet to read data that is provided via the client request. The second argument is a **ServletResponse** object. This enables the servlet to formulate a response for the client.

The call to **setContentType()** establishes the MIME type of the HTTP response. In this program, the MIME type is **text/html**. This indicates that the browser should interpret the content as HTML source code.

Next, the `getWriter()` method obtains a `PrintWriter`. Anything written to this stream is sent to the client as part of the HTTP response. Then `println()` is used to write some simple HTML source code as the HTTP response.

3. Compile the Servlet

Compile the above `HelloServlet.java` file in the command prompt as:

```
javac HelloServlet.java
```

We will get `HelloServlet.class` file. Copy and paste it into `WEB-INF/classes/` directory.

4. Create Deployment Descriptor for your application

Deployment Descriptor(DD) is an XML document that is used by Web Container to run Servlets and JSP pages. DD is used for several important purposes such as:

- Mapping URL to Servlet class.
- Initializing parameters.
- Defining Error page.
- Security roles.
- Declaring tag libraries.

Save this file as `web.xml` into the `WEB-INF` directory.

```
<web-app>

<servlet>
  <servlet-name>hello</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>hello</servlet-name>
  <url-pattern>/hello</url-pattern>
</servlet-mapping>

</web-app>
```

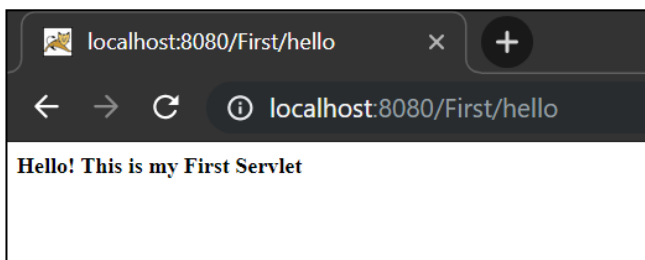
5. Start the server and deploy the application

Start the Apache Tomcat Server before executing the servlet.

Start the web browser and enter the URL as shown below:

```
http://localhost:8080/First/hello
or
http://127.0.0.1: 8080/First/hello
```

We will observe the output as:



10.2.5. The Servlet API

We need to use Servlet API (Application Programming Interface) to create servlets. Two packages contain the classes and interfaces that are required to build servlets. These are ***javax.servlet*** and ***javax.servlet.http***. They constitute the Servlet API. The ***javax.servlet*** and ***javax.servlet.http*** packages represent interfaces and classes for servlet API.

- The ***javax.servlet*** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.
- The ***javax.servlet.http*** package contains interfaces and classes that are responsible for http requests only.

A. The javax.servlet Package

The ***javax.servlet*** package contains a number of interfaces and classes that establish the framework in which servlets operate. The classes and interface in this package are protocol independent.

The following table summarizes the core interfaces that are provided in this package. The most significant of these is ***Servlet***. All servlets must implement this interface or extend a class that implements the interface. The ***ServletRequest*** and ***ServletResponse*** interfaces are also very important.

1.Interface in javax.servlet Package:

S.No.	Interfaces	Description
1.	Servlet	Declares life cycle methods that all servlets must implement.
2.	ServletConfig	Allows servlets to get initialization parameters
3.	ServletContext	Allows servlets to communicate with its servlet container. Enables servlets to log events and access information about their environment.
4.	ServletRequest	Provides client request information to a servlet. Used to read data from a client request.
5.	ServletResponse	Assist a servlet in sending a response to the client. Used to write data to a client response.

Table: Some important Interfaces in javax.servlet package

2.Classes in javax.servlet Package:

The following table summarizes the core classes viz. provided in the ***javax.servlet*** package:

S.No.	Classes	Description
1.	GenericServlet	Provides a basic implementation of the Servlet interface for protocol independent servlets. Implements the Servlet and ServletConfig interfaces.
2.	ServletInputStream	Provides an input stream for reading binary data from a client request.
3.	ServletOutputStream	Provides an output stream for sending binary data to the client.
4.	ServletException	Defines a general exception, a servlet can throw when it encounters difficulty. Indicates a servlet error occurred.
5.	UnavailableException	Indicates that a servlet is not available to service client request.

Table: Some important classes in javax.servlet package

A.1.1. The Servlet Interface

All servlets must implement the *Servlet interface*. It defines the basic structure of a servlet. It is the interface that container use to reference servlets. All servlets implement this interface, either directly or indirectly by extending either the `GenericServlet` or `HttpServlet` class which implements the `Servlet` interface.

It declares the *init()*, *service()*, and *destroy()* methods that are called by the server during the *life cycle of a servlet*.

A method is also provided that allows a servlet to obtain any initialization parameters. The *getServletConfig()* method is called by the servlet to obtain initialization parameters. A servlet developer overrides the *getServletInfo()* method to provide a string with useful information (for example, author, version, date, copyright). The server invokes this method also.

A.1.2. The ServletConfig Interface

The `ServletConfig` interface is implemented by the server. Servers use `ServletConfig` object to pass configuration information to servlets. The configuration information contains initialization parameters, the name of the servlet and a `ServletContext` object which gives servlet information about the container.

The methods declared in this interface are:

S.No.	Methods	Description
1.	<code>String getInitParameter (String name)</code>	It returns a <code>String</code> containing the value of a named initialization parameter or null if specified parameter does not exist.
2.	<code>String getServletName()</code>	It returns the name assigned to a servlet in its deployment descriptor. If no name is specified, this returns the servlet class name instead.
3.	<code>ServletContext getServletContext()</code>	It returns a reference to the <code>ServletContext</code> object for the associated servlet, allowing interaction with the server.
4.	<code>Enumeration getInitParameterName()</code>	It returns the name of the servlet's initialization parameters as an enumeration of <code>String</code> objects or an empty <code>Enumeration</code> if no initialization parameters are specified.

A.1.3. The ServletContext Interface

The `ServletContext` interface provides a means for servlets to communicate with its servlet container. This communication includes finding path information, accessing other servlets running on the server, writing to the server log, getting MIME type of a file and so on. There is one context per web application per Java Virtual Machine.

The `ServletContext` object is a container within the `ServletConfig` object which the web server provides to the servlet when the servlet is initialized .

The `ServletContext` interface specifies the following methods:

S.No.	Methods	Description
1.	<code>Object setAttribute(String name)</code>	It sets the servlet container attribute with the given name or null if the attribute doesnot exist.
2.	<code>String getMimeType(String fileName)</code>	It returns the MIME (Multi-Purpose Internet Mail Extensions) type of a specified file or null if it is not known.
3.	<code>String getRealPath(String vpath)</code>	It returns the real path (on the server file system) that corresponds to the virtual path vpath.
4.	<code>String getServerInfo()</code>	It returns the name and version of the servlet container separated by a forward slash (/).
5.	<code>void setAttribute(String name, Object obj)</code>	It binds a specified object to a given attribute name in this servlet context.

6.	void log (String msg)	It writes the specified message to a servlet log file usually an event log.
7.	void log(String msg, Throwable t)	It writes the specified message msg and a stack trace for a specified Throwable exception t to the servlet log file.

A.1.4. ServletRequest Interface

The ServletRequest interface defines an object that encapsulates information about a single client request. The servlet container creates a ServletRequest object and passes it as an argument to the servlet's service () method. Data provided by the object generally includes parameter names and values, implementation specific attributes and an input stream for reading binary data from the request body. Some of the methods specified in it are given below :

S.No.	Methods	Description
1.	Object getAttribute(String name)	It returns the value of the named attribute as an object, or null if the named attribute does not exist.
2.	Enumeration getAttributeNames()	It returns an enumeration of all attributes contained in the request.
3.	int getContentLength()	It returns the length (in bytes) of the content being sent via the input stream or -1 if the length is not known.
4.	String getContentType()	It request the MIME type of the body of the request or null if the type is not known.
5.	ServletInputStream getInputStream()	It retrieves the body of the request as binary data using ServletInputStream object.
6.	String getparameter (String name)	It returns the value of the specified parameter or null if the parameter does not exists or without a value.
7.	Enumeration getParameterNames()	It returns all the parameter names as enumeration of String objects.
8.	String [] getParameterValues(String name)	It returns an array of String objects containing all the values of the specified parameter or null if the parameter does not exist.
9.	String getprotocol()	It returns the name and version of the protocol used by the request.
10.	String getParameterAddr()	It returns the IP address of the client that sent the request.
11.	String getRemoteHost()	It returns the name of the client that send the request.
12.	String getScheme()	It returns the name and the scheme used to make the request. For example : http, ftp,https etc.
13.	String getServerName()	It returns the name of the server that receives the request.
14.	int getServerport()	It returns the port number on which the request was received.
15.	BufferedReader getReader()	It retrieves the body of the request as character data.

A.1.5. The Servlet Response Interface

The ServletResponse interface is the response counterpart of the ServletRequest object. It defines an object to assist a servlet in sending MIME encoded data back to the client. The servlet container

create a `ServletResponse` object and passes it as an argument to the servlet's `service ()` method. The *ServletResponse* interface enables a servlet to formulate a response for a client.

Some of the most commonly used methods of this interface are as follows:

S.No.	Methods	Description
1.	<code>void setContentType(String type)</code>	It sets the MIME type of the response being sent to the client. For example, in case of HTML, the MIME type should be set to "text/HTML".
2.	<code>void setContentLength(int len)</code>	It sets the length of the content being returned by the server. In HTTP servlets, this method sets the HTTP content -length header.
3.	<code>ServletOutputStream getOutputStream()</code>	It returns a <code>ServletOutputStream</code> object than can be used for writing binary data into the response.
4.	<code>PrintWriter getWriter()</code>	It returns a <code>PrintWriter</code> object that can send character text in the response.
5.	<code>String getCharacterEncoding()</code>	It returns the name of the charset used for the MIME body sent in the response.

A.2.1. The GenericServlet CLASS

The `GenericServlet` class provides a basic implementation of the `Servlet` interface. This is an abstract class and all subclasses should implement the `service ()` method. This class implements both `Servlet` as well as `ServletConfig` interface. It has the following methods in addition to those declared in `Servlet` and `ServletConfig` interfaces.

The *GenericServlet* class provides implementations of the basic life cycle methods for a servlet. *GenericServlet* implements the *Servlet* and *ServletConfig* interfaces. In addition, a method to append a string to the server log file is available. The signatures of this method are shown here:

S.No.	Methods	Description
1.	<code>void init ()</code>	It is similar to <code>init (ServletConfig config)</code> . This method is provided as a convenience so that servlet developers do not have to worry about storing the <code>ServletConfig</code> object.
2.	<code>void log (String msg)</code>	It writes the specified message <code>msg</code> to a servlet log file.
3.	<code>void log(String msg, Throwable t)</code>	It writes the name of the servlet, the specified message <code>msg</code> and the stack trace <code>t</code> to the servlet log file

A.2.2 ServletInputStream CLASS

The `ServletInputStream` class extends `InputStream`. It provides input stream for reading binary data from a client request. A servlet obtains a `ServletInputStream` object from the `getInputStream ()` method of `ServletRequest`. It defines the default constructor which does nothing.

A.2.3 ServletOutputStream CLASS

The `ServletOutputStream` class extends `OutputStream`. It provides an output stream for sending binary data back to the client. A servlet obtains a `ServletOutputStream` object from the `getOutputStream ()` method of `ServletResponse`. It also has a default constructor that does nothing. In addition, it also includes a range of `print ()` and `println ()` methods for sending text or HTML.

It takes the following form

- `void print(type v)`
- `void println(type v)`

Here, `type` specifies the datatype `int`, `char`, `float`, `long`, `String`, `double`, `boolean` and `v` specifies the data to be written.

A.2.4 ServletException CLASS

The `javax.Servlet` defines two exception classes `ServletException` and `UnavailableException`. The `ServletException` is a generic exception which can be thrown by servlets encountering difficulties. `UnavailableException` is a special type of servlet exception which extends the `ServletException` class. The purpose of this class is to indicate to the servlet container that the servlet is either temporarily or permanently unavailable to service client requests.

3. Reading Servlet Parameters

The ***ServletRequest*** interface includes methods that allow us to read the names and values of parameters that are included in a client request.

Let us consider the below example that contains two files. A web page is defined in *index.html*, and a servlet is defined in *ReadingServletParameterDemo.java*.

The HTML source code for *index.html* is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is *Employee* and the other is *Phone*. There is also a *submit* button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

Saved as: *index.html* in a folder named *ReadingServletParameterDemo* in *webapps* (i.e. C:\Program Files (x86)\Apache Software Foundation\Tomcat 9.0\webapps\ReadingServletParameterDemo)

```
<html>
<body>
    <center>
    <form name="Form1" method="post"
    action="http://localhost:8080/ReadingServletParameterDemo/parameter">
    <table>
    <tr>
    <td><B>Employee</td>
    <td><input type="text" name="e" size="25" value=""></td>
    </tr>
    <tr>
    <td><B>Phone</td>
    <td><input type="text" name="p" size="25" value=""></td>
    </tr>
    </table>
    <input type="submit" value="Submit">
</body>
</html>
```

The source code for *ReadingServletParameterDemo.java* is shown in the following listing. The ***service()*** method is overridden to process client requests. The ***getParameterNames()*** method returns an enumeration of the parameter names. These are processed in a loop. We can see that the parameter name and value are output to the client. The parameter value is obtained via the ***getParameter()*** method.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
public class ReadingServletParameterDemo extends GenericServlet {
    public void service(ServletRequest request, ServletResponse response)
    throws ServletException, IOException {
        // Get print writer.
        PrintWriter pw = response.getWriter();
```

```

        // Get enumeration of parameter names.
        Enumeration e = request.getParameterNames();
        // Display parameter names and values.
        while(e.hasMoreElements()) {
            String pname = (String)e.nextElement();
            pw.print(pname + " = ");
            String pvalue = request.getParameter(pname);
            pw.println(pvalue);
        }
        pw.close();
    }
}

```

The **web.xml** file is as:

```

<web-app>

<servlet>
<servlet-name>demo2</servlet-name>
<servlet-class>ReadingServletParameterDemo</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>demo2</servlet-name>
<url-pattern>/parameter</url-pattern>
</servlet-mapping>

</web-app>

```

Step 1:

Lets us first create a folder **ReadingServletParameterDemo** in the **webapps** directory of Tomcat (i.e. C:\Program Files (x86)\Apache Software Foundation\Tomcat 9.0\webapps).

Step 2:

Create a **index.html** file inside the folder with above mentioned code.

Step 3:

Write a servlet program as given above and save it as **ReadingServletParameterDemo.java** inside **src** folder which is inside the **ReadingServletParameterDemo** folder. (i.e. C:\Program Files (x86)\Apache Software Foundation\Tomcat 9.0\webapps\ReadingServletParameterDemo\src)

Step 4:

Compile the above java file in command prompt and put the **ReadingServletParameterDemo.class** file inside the WEB-INF/classes directory (i.e. C:\Program Files (x86)\Apache Software Foundation\Tomcat 9.0\webapps\ReadingServletParameterDemo\WEB-INF\classes)

Step 5:

Create a web.xml file as given above in the WEB-INF directory (i.e. C:\Program Files (x86)\Apache Software Foundation\Tomcat 9.0\webapps\ ReadingServletParameterDemoDemo \WEB-INF).

Step 6:

Start Tomcat.

Step 7:

Open a web browser and type <http://localhost:8080/ReadingServletParameterDemo/index.html>, you should get as:

Employee

Phone

Employee Shiva

Phone 12345

Step 8:

Supply the information and submit it. You should get a message as:

e = Shiva
p = 12345

Assignment:

Write a code for following example using Servlet:

Username

Password

B. 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.

1.Interface in javax.servlet.http Package:

Interface	Description
HttpServletRequest	Enables servlets to read data from an HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and written.
HttpSessionBindingListener	Notifies an object that it is bound to or unbound from a session.

Table: Some important Interfaces in javax.servlet.http package

2.Classes in javax.servlet.http Package:

Class	Description
Cookie	Allows state information to be stored on a client machine.
HttpServlet	Provides methods to handle HTTP requests and responses.
HttpSessionEvent	Encapsulates a session-changed event.
HttpSessionBindingEvent	Indicates when a listener is bound to or unbound from a session value, or that a session attribute changed.

Table: Some important classes in javax.servlet.http package

B.1.1. The HttpServletRequest Interface

HttpServletRequest extends javax.servlet.ServletException and provides a number of methods that make it easy to access specific information related to an HTTP request. The *HttpServletRequest* interface enables a servlet to obtain information about a client request.

Several of its methods are shown in following table:

Method	Description
String getAuthType()	Returns authentication scheme.
Cookie[] getCookies()	Returns an array of the cookies in this request.
long getDateHeader(String field)	Returns the value of the date header field named <i>field</i> .
String getHeader(String field)	Returns the value of the header field named <i>field</i> .
Enumeration getHeaderNames()	Returns an enumeration of the header names.
int getIntHeader(String field)	Returns the int equivalent of the header field named <i>field</i> .
String getMethod()	Returns the HTTP method for this request.
String getPathInfo()	Returns any path information that is located after the servlet path and before a query string of the URL.
String getPathTranslated()	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
String getQueryString()	Returns any query string in the URL.
String getRemoteUser()	Returns the name of the user who issued this request.
String getRequestedSessionId()	Returns the ID of the session.
String getRequestURI()	Returns the URI.
StringBuffer getRequestURL()	Returns the URL.
String getServletPath()	Returns that part of the URL that identifies the servlet.
HttpSession getSession()	Returns the session for this request. If a session does not exist, one is created and then returned.
HttpSession getSession(boolean new)	If <i>new</i> is true and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
boolean isRequestedSessionIdFromCookie()	Returns true if a cookie contains the session ID. Otherwise, returns false .
boolean isRequestedSessionIdFromURL()	Returns true if the URL contains the session ID. Otherwise, returns false .
boolean isRequestedSessionIdValid()	Returns true if the requested session ID is valid in the current session context.

B.1.2. The HttpServletResponse Interface

The *HttpServletResponse* interface enables a servlet to formulate an HTTP response to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response.

For example, SC_OK indicates that the HTTP request succeeded, and SC_NOT_FOUND indicates that the requested resource is not available.

Several methods of this interface are summarized in table below.

Method	Description
void addCookie(Cookie cookie)	Adds <i>cookie</i> to the HTTP response.
boolean containsHeader(String field)	Returns true if the HTTP response header contains a field named <i>field</i> .
String encodeURL(String url)	Determines if the session ID must be encoded in the URL identified as <i>url</i> . If so, returns the modified version of <i>url</i> . Otherwise, returns <i>url</i> . All URLs generated by a servlet should be processed by this method.
String encodeRedirectURL(String url)	Determines if the session ID must be encoded in the URL identified as <i>url</i> . If so, returns the modified version of <i>url</i> . Otherwise, returns <i>url</i> . All URLs passed to sendRedirect() should be processed by this method.

Method	Description
<code>void sendError(int c)</code> throws <code>IOException</code>	Sends the error code <i>c</i> to the client.
<code>void sendError(int c, String s)</code> throws <code>IOException</code>	Sends the error code <i>c</i> and message <i>s</i> to the client.
<code>void sendRedirect(String url)</code> throws <code>IOException</code>	Redirects the client to <i>url</i> .
<code>void setDateHeader(String field, long msec)</code>	Adds <i>field</i> to the header with date value equal to <i>msec</i> (milliseconds since midnight, January 1, 1970, GMT).
<code>void setHeader(String field, String value)</code>	Adds <i>field</i> to the header with value equal to <i>value</i> .
<code>void setIntHeader(String field, int value)</code>	Adds <i>field</i> to the header with value equal to <i>value</i> .
<code>void setStatus(int code)</code>	Sets the status code for this response to <i>code</i> .

B.1.3. The HttpSession Interface

The `HttpSession` interface is the core of the session tracking functionality introduced in Version 2.0 of the Servlet API. The `HttpSession` interface enables a servlet to read and write the state information that is associated with an HTTP session.

A servlet obtains an `HttpSession` objects from the `getSession()` method of `HttpServletRequest`. The `putValue()` and `removeValue()` methods bind Java objects to a particular session.

Several of its methods are summarized in Table below. All of these methods throw an `IllegalStateException` if the session has already been invalidated.

Method	Description
<code>Object getAttribute(String attr)</code>	Returns the value associated with the name passed in <i>attr</i> . Returns null if <i>attr</i> is not found.
<code>Enumeration getAttributeNames()</code>	Returns an enumeration of the attribute names associated with the session.
<code>long getCreationTime()</code>	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when this session was created.
<code>String getId()</code>	Returns the session ID.
<code>long getLastAccessedTime()</code>	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request for this session.
<code>void invalidate()</code>	Invalidates this session and removes it from the context.
<code>boolean isNew()</code>	Returns true if the server created the session and it has not yet been accessed by the client.
<code>void removeAttribute(String attr)</code>	Removes the attribute specified by <i>attr</i> from the session.
<code>void setAttribute(String attr, Object val)</code>	Associates the value passed in <i>val</i> with the attribute name passed in <i>attr</i> .

B.1.4. The HttpSessionBindingEvent Interface

An `HttpSessionBindingEvent` is passed to the appropriate method of an `HttpSessionBindingListener` when an object is bound to or unbound from an `HttpSession`. The `getName()` method returns the name to which the bound object has been assigned, and the `getSession()` method provides a reference to the session the object is being bound to.

The `HttpSessionBindingListener` interface is implemented by objects that need to be notified when they are bound to or unbound from an HTTP session. The methods that are invoked when an object is bound or unbound are:

```
void valueBound(HttpSessionBindingEvent e)
void valueUnbound(HttpSessionBindingEvent e)
```

Here, *e* is the event object that describes the binding.

B.2.1. The Cookie Class

The Cookie class encapsulates a cookie. A **cookie** is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assume that a user visits an online store. A cookie can save the user's name, address, and other information.

The user does not need to enter this data each time he or she visits the store. A servlet can write a cookie to a user's machine via the ***addCookie()*** method of the ***HttpServletResponse*** interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser.

The names and values of cookies are stored on the user's machine. Some of the information that is saved for each cookie includes the following:

- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user's machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends. Otherwise, the cookie is saved in a file on the user's machine.

The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the Web server. Otherwise, it is not. There is one constructor for Cookie. It has the signature shown here:

Cookie(String name, String value).

Here, the name and value of the cookie are supplied as arguments to the constructor.

Method	Description
<code>Object clone()</code>	Returns a copy of this object.
<code>String getComment()</code>	Returns the comment.
<code>String getDomain()</code>	Returns the domain.
<code>int getMaxAge()</code>	Returns the maximum age (in seconds).
<code>String getName()</code>	Returns the name.
<code>String getPath()</code>	Returns the path.
<code>boolean getSecure()</code>	Returns true if the cookie is secure. Otherwise, returns false .
<code>String getValue()</code>	Returns the value.
<code>int getVersion()</code>	Returns the version.
<code>void setComment(String c)</code>	Sets the comment to <i>c</i> .
<code>void setDomain(String d)</code>	Sets the domain to <i>d</i> .
<code>void setMaxAge(int secs)</code>	Sets the maximum age of the cookie to <i>secs</i> . This is the number of seconds after which the cookie is deleted.
<code>void setPath(String p)</code>	Sets the path to <i>p</i> .
<code>void setSecure(boolean secure)</code>	Sets the security flag to <i>secure</i> .
<code>void setValue(String v)</code>	Sets the value to <i>v</i> .
<code>void setVersion(int v)</code>	Sets the version to <i>v</i> .

B.2.2. The `HttpServlet` Class

The `HttpServlet` class extends `GenericServlet`. It is commonly used when developing servlets that receive and process HTTP requests. The methods defined by the `HttpServlet` class are summarized in Table below:

Method	Description
<code>void doDelete(HttpServletRequest req, HttpServletResponse res)</code> throws <code>IOException</code> , <code>ServletException</code>	Handles an HTTP DELETE request.
<code>void doGet(HttpServletRequest req, HttpServletResponse res)</code> throws <code>IOException</code> , <code>ServletException</code>	Handles an HTTP GET request.
<code>void doHead(HttpServletRequest req, HttpServletResponse res)</code> throws <code>IOException</code> , <code>ServletException</code>	Handles an HTTP HEAD request.
<code>void doOptions(HttpServletRequest req, HttpServletResponse res)</code> throws <code>IOException</code> , <code>ServletException</code>	Handles an HTTP OPTIONS request.
<code>void doPost(HttpServletRequest req, HttpServletResponse res)</code> throws <code>IOException</code> , <code>ServletException</code>	Handles an HTTP POST request.
<code>void doPut(HttpServletRequest req, HttpServletResponse res)</code> throws <code>IOException</code> , <code>ServletException</code>	Handles an HTTP PUT request.
<code>void doTrace(HttpServletRequest req, HttpServletResponse res)</code> throws <code>IOException</code> , <code>ServletException</code>	Handles an HTTP TRACE request.
<code>long getLastModified(HttpServletRequest req)</code>	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
<code>void service(HttpServletRequest req, HttpServletResponse res)</code> throws <code>IOException</code> , <code>ServletException</code>	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

B.2.3. The `HttpSessionEvent` Class

// will be dealt in session management

B.2.4. The `HttpSessionBindingEvent` Class

// will be dealt in session management

10.3. HTTP request handling, Session Management

The *HttpServlet* class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are *doDelete()*, *doGet()*, *doHead()*, *doOptions()*, *doPost()*, *doPut()*, and *doTrace()*. The GET and POST requests are commonly used when handling form input.

10.3.1. Handling HTTP GET Requests

The *doGet()* method is invoked by server through *service()* method to handle a HTTP GET request. This method also handles HTTP HEAD request automatically as HEAD request is nothing but a GET request having no body in the code for response and only includes request header fields. To understand the working of *doGet()* method, let us consider a sample program to define a servlet for handling the HTTP GET request.

Step 1: Lets us first create a folder *GetRequestDemo* in the *webapps* directory of Tomcat (i.e. C:\Program Files (x86)\Apache Software Foundation\Tomcat 9.0\webapps).

Step 2: Create a *chooser.html* file inside the folder with below code:

```
<html>
<body>
    <center>
    <form name="Form1"action="http://localhost:8080/GetRequestDemo/colorchooser">
        <B>Color:</B>
        <select name="color" size="1">
            <option value="Red">Red</option>
            <option value="Green">Green</option>
            <option value="Blue">Blue</option>
        </select>
        <br><br>
        <input type=submit value="Submit">
    </form>
    </body>
</html>
```

Step 3: Write a servlet program and save it as *GetRequestDemo.java* inside *src* folder which is inside the *GetRequestDemo* folder. (i.e. C:\Program Files (x86)\Apache Software Foundation\Tomcat 9.0\webapps\GetRequestDemo\src)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class GetRequestDemo extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.close();
    }
}
```

In the above servlet program, the *doGet()* method is overridden to process any HTTP GET requests that are sent to this servlet. It uses the *getParameter()* method of *HttpServletRequest* to obtain the selection that was made by the user. A response is then formulated.

Step 4: Compile the above java file in command prompt and put the `GetRequestDemo.class` file inside the `WEB-INF/classes` directory (i.e. `C:\Program Files (x86)\Apache Software Foundation\Tomcat 9.0\webapps\GetRequestDemo\WEB-INF\classes`)

Step 5: Create a `web.xml` file as below in the `WEB-INF` directory (i.e. `C:\Program Files (x86)\Apache Software Foundation\Tomcat 9.0\webapps\GetRequestDemo\WEB-INF`).

```
<web-app>

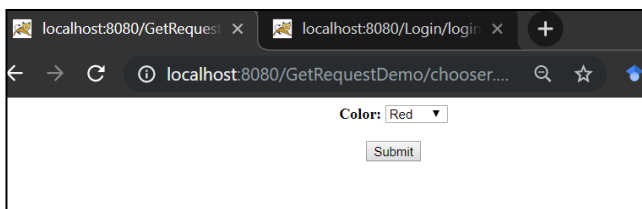
<servlet>
<servlet-name>demo</servlet-name>
<servlet-class>GetRequestDemo</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>demo</servlet-name>
<url-pattern>/colorchooser</url-pattern>
</servlet-mapping>

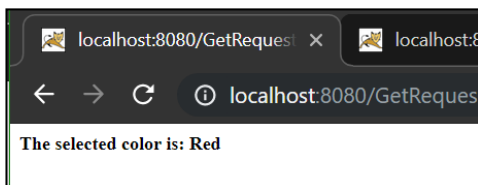
</web-app>
```

Step 6: Start Tomcat.

Step 7: Open a web browser and type <http://localhost:8080/GetRequestDemo/chooser.html> , you should get as:



Step 8: Select the color (say you selected Red) and submit it. You should get a message as:



10.3.2. Handling HTTPPOST Requests

Like *doGet()* method, the *doPost()* method is invoked by server through *service()* method to handle HTTP POST request. The *doPost()* method is used when large amount of data is required to be passed to the server which is not possible with the help of *doGet()* method.

In *doGet()* method, parameters are appended to the URL whereas, in *doPost()* method parameters are sent in separate line in the HTTP request body. The *doGet()* method is mostly used when some information is to be retrieved from the server and the *doPost()* method is used when data is to be updated on server or data is to be submitted to the server. To understand the working of *doPost()* method, let us consider a sample program to define a servlet for handling the HTTP POST request.

The steps are all as we discussed above in the HTTPGET Request.

The servlet program is:

```
//saved as GetPostDemo.java

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetPostDemo extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.close();
    }
}
```

The html file is: saved as *chooser.html*

```
<html>
<body>
    <center>
        <form name="Form1" method="post"
action="http://localhost:8080/GetPostDemo/colorchooser">
            <B>Color:</B>
            <select name="color" size="1">
                <option value="Red">Red</option>
                <option value="Green">Green</option>
                <option value="Blue">Blue</option>
            </select>
            <br><br>
            <input type="submit" value="Submit">
        </form>
    </body>
</html>
```

The *web.xml* file is as:

```
<web-app>

<servlet>
<servlet-name>demo1</servlet-name>
<servlet-class>GetPostDemo</servlet-class>
```

```

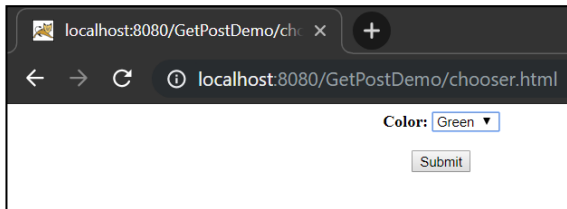
</servlet>

<servlet-mapping>
<servlet-name>demo1</servlet-name>
<url-pattern>/colorchooser</url-pattern>
</servlet-mapping>

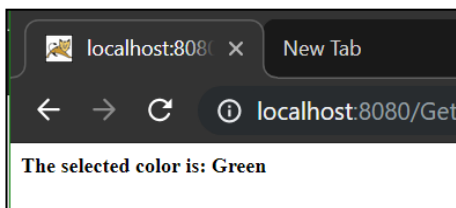
</web-app>

```

When we type <http://localhost:8080/GetPostDemo/chooser.html> , we get as:



Upon selecting the color (say we selected Green) and submitted it. we should get a message as:



10.3.3. Session Tracking

Session simply means a particular interval of time. Session Tracking is a way to maintain state (data) of an user. It is also known as session management in servlet.

Http protocol is a stateless. It means each request is considered as the new request. 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.

A session can be created via the *getSession()* method of *HttpServletRequest*. An *HttpSession* object is returned. This object can store a set of bindings that associate names with objects. The *setAttribute()*, *getAttribute()*, *getAttributeNames()*, and *removeAttribute()* methods of *HttpSession* manage these bindings. It is important to note that session state is shared among all the servlets that are associated with a particular client.

The following servlet illustrates how to use session state. The *getSession()* method gets the current session. A new session is created if one does not already exist. The *getAttribute()* method is called to obtain the object that is bound to the name "date". That objects is a Date object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A Date object encapsulating the current date and time is then created. The *setAttribute()* method is called to bind the name "date" to this object.

```

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {

```

```

// Get the HttpSession object.
HttpSession hs = request.getSession(true);

// Get writer.
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.print("<B>");

// Display date/time of last access.
Date date = (Date)hs.getAttribute("date");
if(date != null) {
    pw.print("Last access: " + date + "<br>");
}

// Display current date/time.
date = new Date();
hs.setAttribute("date", date);
pw.println("Current date: " + date);
}

```

When we first request this servlet, the browser displays one line with the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.

10.4. JSP life cycle, Writing JSP pages

10.4.1. JSP

- JSP technology is used to create web application just like Servlet technology. It can be thought of as an extension to Servlet because it provides more functionality than servlet such as expression language, JSTL, etc.
- Java Server Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications.
- 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 `%>`.
- A **JSP** component is a type of *Java servlet* that is designed to fulfill the role of a user interface for a Java web application.
- Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.
- Using JSP, we can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.
- JSP tags can be used for a variety of purposes, such as retrieving information from a database or *registering user preferences, accessing JavaBeans components, passing control between pages and sharing information between requests, pages* etc.

10.4.2. Advantages of JSP over Servlet

There are many advantages of JSP over the Servlet. They are as follows:

1. Extension to Servlet

JSP technology is the extension to Servlet technology. We can use all the features of the Servlet in JSP. In addition to, we can use implicit objects, predefined tags, expression language and Custom tags in JSP, that makes JSP development easy.

2. Easy to maintain

JSP can be easily managed because we can easily separate our business logic with presentation logic. In Servlet technology, we mix our business logic with the presentation logic.

3. Fast Development: No need to recompile and redeploy

If JSP page is modified, we don't need to recompile and redeploy the project. The Servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

4. Less code than Servlet

In JSP, we can use many tags such as action tags, JSTL, custom tags, etc. that reduces the code. Moreover, we can use EL, implicit objects, etc.

10.4.2. JSP vs Different Technologies

JSP is one of the most widely used language over the web. Following is the list of other advantages of using JSP over other technologies:

1. JSP vs. Active Server Pages (ASP)

The advantages of JSP are twofold. First, the dynamic part is written in Java, not Visual Basic or other MS specific language, so it is more powerful and easier to use. Second, it is portable to other operating systems and non-Microsoft Web servers.

2. JSP vs. Pure Servlets

It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.

3. JSP vs. Server-Side Includes (SSI)

SSI is really only intended for simple inclusions, not for "real" programs that use form data, make database connections, and the like.

4. JSP vs. JavaScript

JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.

5. JSP vs. Static HTML

Regular HTML, of course, cannot contain dynamic information.

10.4.2. JSP Life Cycle

A JSP life cycle can be defined as the entire process from its creation till the destruction which is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet. The following are the paths followed by a JSP

1. Compilation
2. Initialization
3. Execution
4. Cleanup

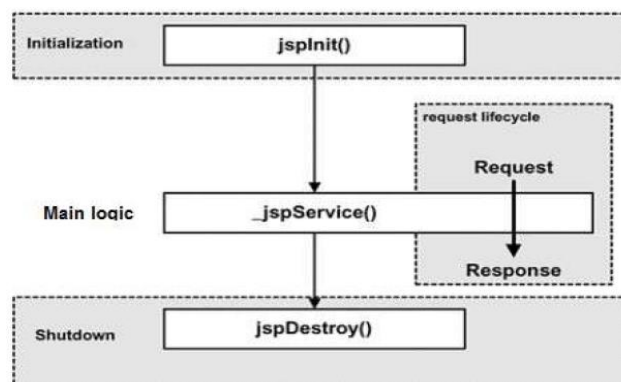


Figure: JSP Life Cycle

1. Compilation

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page. The compilation process involves three steps:

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

2. Initialization

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

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

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

3. Execution

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed. Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the *_jspService()* method in the JSP. The *jspService()* method takes an **HttpServletRequest** and an **HttpServletResponse** as its parameters as follows:

```
void _jspService(HttpServletRequest request,
HttpServletResponse response)
{
    // Service handling code...
}
```

The *jspService()* method of a JSP is invoked once per a request and is responsible for generating the response for that request and this method is also

4. Cleanup

The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container. The *jspDestroy()* method is the JSP equivalent of the destroy method for servlets. We need to override *jspDestroy* to perform any cleanup, such as releasing database connections or closing open files.

The *jspDestroy()* method has the following form:

```
public void jspDestroy()
{
    // Your cleanup code goes here.
}
```

10.4.3. JSP Syntax

1. Declaration Tag :

It is used to declare variables and methods

Syntax:-

```
<%! Dec var %>
```

Example:-

```
<%! int var=10; %>
```


Eg:

```
<%!
private int count=0;
private void incrementCount()
{
    count++;
}
%>
```

2. JSP Expression :

It evaluates and convert the expression to a string. We can also access variables defined in declarations with expression. The syntax to embed an expression is as follows:

Syntax:-

```
<%= expression %>
```

Example:-

```
<% num1 = num1+num2 %>
```

Expressions are embedded directly into the HTML. The Web browser will display the value of the expression in place of the tag. For example, we can output the value of the **count** variable in bold type with the following piece of HTML:

```
The value of count is <b> <%= count %> </b>
```

3. Java Scriptlets :

It allows us to add any number of JAVA code, variables and expressions. Blocks of Java code can be embedded in a *scriptlet*.

Syntax:-

```
<% java code %>
```

If we wish to output HTML within a *scriptlet*, then this is done using *out.println()*, which is used in the same manner as *System.out.println()*. The variable *out* is already defined for us and is of type *javax.servlet.jsp.JspWriter*. Also note that *System.out.println()* will output to the console, which is useful for debugging purposes, while *out.println()* will output to the browser. The following *scriptlet* invokes the *incrementCount()* method and then outputs the value in count :

```
<%
out.println("The counter's value is " + count + "<br />");
incrementCount();
%>
```

4. JAVA Comments :

It contains the text that is added for information which has to be ignored.

Syntax:-

```
<% -- JSP Comments %>
```

5. JSP Directives

Finally, let us introduce one more JSP tag, the *directive*. In general terms, directives instruct the compiler how to process a JSP program. Examples include the definition of our own tags, including the source code of other files, and importing packages. The syntax for directives is as follows:

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

10.4.3. A Simple JSP program

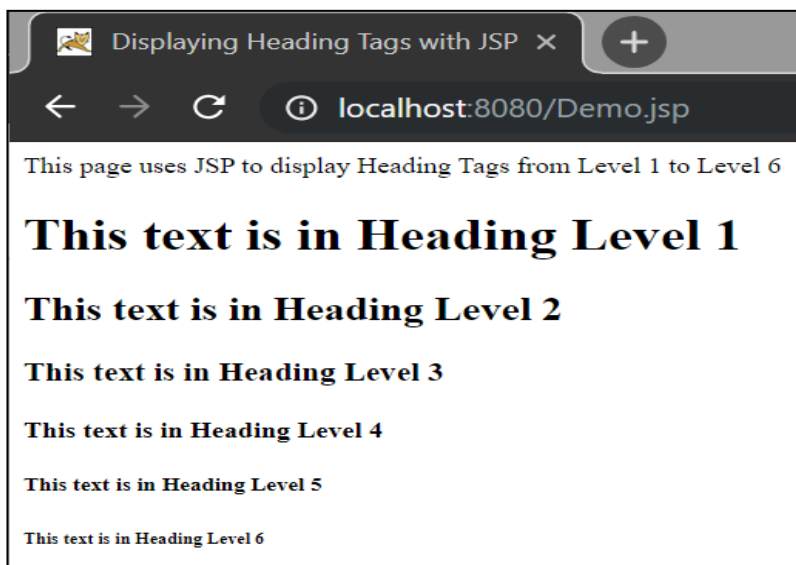
Save the below as **Demo.jsp** and place it into the **root** directory under **web-apps** in Tomcat directory (i.e. C:\Program Files (x86)\Apache Software Foundation\Tomcat 9.0\webapps\ROOT)

```
<html>
<title>
    Displaying Heading Tags with JSP
</title>
<body>

    <%!
    private static final int LASTLEVEL = 6;
    %>

    <p>
    This page uses JSP to display Heading Tags from
    Level 1 to Level <%= LASTLEVEL %>
    </p>
    <%
        int i;
        for (i = 1; i <= LASTLEVEL; i++)
        {
            out.println("<H" + i + ">" +
            "This text is in Heading Level " + i +
            "</H" + i + ">");
        }
    %>
</body>
</html>
```

Now start Tomcat and open web-browser and navigate as: <http://localhost:8080/Demo.jsp>
The output observed is :



10.4.4 Handling Parameters in JSP

To make a JSP page more interactive, we can read and process the data entered in an HTML form.

One way to read these values is to call the *request.getParameter()* method. This method takes a String parameter as input that identifies the name of an HTML form element and returns the value entered by the user for that element on the form.

For example, if there is a *textbox* named *AuthorID*, then we can retrieve the value entered in that textbox with the following *scriptlet* code:

```
String value = request.getParameter("AuthorID");
```

If the user leaves the field blank, then *getParameter* returns an empty string.

Create a new Test.html file and save in the ROOT folder of Tomcat. This Html file uses a *EditURL.jsp* in order to process request.

```
<html>
<head>
    <title>Change Author's URL</title>
</head>
<body>
    <h1>Change Author's URL</h1>
    <p>
        Enter the ID of the author you would like to change
        along with the new URL.
    </p>
    <form ACTION = "EditURL1.jsp" METHOD = POST>
        Author ID:
        <input TYPE = "TEXT" NAME = "AuthorID" VALUE = "" SIZE = "4"
        MAXLENGTH = "4">
        <br/>
        New URL:
        <input TYPE = "TEXT" NAME = "URL"
        VALUE = "http://" SIZE = "40" MAXLENGTH = "200">
        <p>
            <input type=submit value="Submit">
        </p>
    </form>
</body>
</html>
```

Also create another EditURL.jsp file and place it also in the ROOT folder of Tomcat. This JSP program echoes back the data entered by the user in above html. To be careful, the name of the JSP file must match the value supplied for the ACTION tag of the form.

```
<html>
    <title>Edit URL: Echo submitted values</title>
<body>
    <h2>Edit URL</h2>
    <p>
        This version of EditURL.jsp simply echoes back to the
        user the values that were entered in the textboxes.
    </p>
    <%
        String url = request.getParameter("URL");
        String stringID = request.getParameter("AuthorID");
        int author_id = Integer.parseInt(stringID);
```

```

        out.println("The submitted author ID is: " + author_id);
        out.println("<br/>");
        out.println("The submitted URL is: " + url);
        %>
</body>
</html>

```

Result:

Change Author's URL

Enter the ID of the author you would like to change along with the new URL.

Author ID:

New URL:

Edit URL: Echo submitted values

This version of EditURL.jsp simply echoes back to the user the values that were entered in the textboxes.

The submitted author ID is: 555

The submitted URL is: http://merojob.com

Assignment:

Give the HTML to create a form with two elements: a textbox named *FirstName* that holds a maximum of 50 characters, a *Submit* button. The form should submit its data to a JSP program called *Process.jsp* using the POST method. Also, write the *process.jsp* to handle these requests.

10.4.4 JSP Implicit Objects

JSP Implicit Objects are the Java objects that the JSP Container makes available to developers in each page and developer can call them directly without being explicitly declared. JSP Implicit Objects are also called pre-defined variables. JSP supports nine Implicit Objects which are listed below:

Object	Description
request	This is the HttpServletRequest object associated with the request.
response	This is the HttpServletResponse object associated with the response to the client.
out	This is the PrintWriter object used to send output to the client.
session	This is the HttpSession object associated with the request.
application	This is the ServletContext object associated with application context.
config	This is the ServletConfig object associated with the page.
pageContext	This encapsulates use of server-specific features like higher performance JspWriters .
page	This is simply a synonym for this, and is used to call the methods defined by the translated servlet class.
Exception	The Exception object allows the exception data to be accessed by designated JSP.

1. The request object

The request object is an instance of a *javax.servlet.http.HttpServletRequest* object. Each time a client requests a page the JSP engine creates a new object to represent that request. The *request* object provides methods to get HTTP header information including form data, cookies, HTTP methods etc.

2. The *response* object

The *response object* is an instance of a *javax.servlet.http.HttpServletResponse* object. Just as the server creates the request object, it also creates an object to represent the response to the client. The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes etc.

3. The *out* object

The *out* implicit object is an instance of a *javax.servlet.jsp.JspWriter* object and is used to send content in a response. The initial *JspWriter* object is instantiated differently depending on whether the page is buffered or not. Buffering can be easily turned off by using the `buffered=false` attribute of the page directive. The *JspWriter* object contains most of the same methods as the *java.io.PrintWriter* class. However, *JspWriter* has some additional methods designed to deal with buffering. Unlike the *PrintWriter* object, *JspWriter* throws *IOExceptions*.

4. The *session* object

The *session object* is an instance of *javax.servlet.http.HttpSession* and behaves exactly the same way that session objects behave under Java Servlets. The session object is used to track client session between client requests.

5. The *application* object

The *application object* is direct wrapper around the *ServletContext* object for the generated Servlet and in reality an instance of a *javax.servlet.ServletContext* object. This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the *jspDestroy()* method.

6. The *config* object

The *config object* is an instantiation of *javax.servlet.ServletConfig* and is a direct wrapper around the *ServletConfig* object for the generated servlet. This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc.

7. The *pageContext* object

The *pageContext* object is an instance of a *javax.servlet.jsp.PageContext* object. The *pageContext* object is used to represent the entire JSP page. This object is intended as a means to access information about the page while avoiding most of the implementation details. This object stores references to the request and response objects for each request. The *application*, *config*, *session*, and *out* objects are derived by accessing attributes of this object.

8. *Page* object

The *Page* object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page. The page object is really a direct synonym for the **this** object.

9. The *exception* object

The *exception* object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

10.5. Introduction to JSP tag library

JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates core functionality common to many JSP applications. JSTL has support for *common*, *structural tasks* such as *iteration and conditionals*, *tags for manipulating XML documents*, *internationalization tags*, and *SQL tags*. It also provides a framework for integrating existing custom tags with JSTL tags.

The JSTL tags can be classified, according to their functions, into following JSTL tag library groups that can be used when creating a JSP page:

1. Core Tags
2. Formatting tags
3. SQL tags
4. XML tags
5. JSTL Functions

1. Core Tags

The core group of tags is the most frequently used JSTL tags. Following is the syntax to include JSTL Core library in our JSP:

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

Tag	Description
<u><c.out></u>	Like <code><%= ... ></code> , but for expressions.
<u><c.set></u>	Sets the result of an expression evaluation in a 'scope'
<u><c.remove></u>	Removes a scoped variable (from a particular scope, if specified).
<u><c.catch></u>	Catches any Throwable that occurs in its body and optionally exposes it.
<u><c.if></u>	Simple conditional tag which evaluates its body if the supplied condition is true.
<u><c.choose></u>	Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <code><when></code> and <code><otherwise></code>
<u><c.when></u>	Subtag of <code><choose></code> that includes its body if its condition evaluates to 'true'.
<u><c.otherwise></u>	Subtag of <code><choose></code> that follows <code><when></code> tags and runs only if all of the prior conditions evaluated to 'false'.
<u><c.import></u>	Retrieves an absolute or relative URL and exposes its contents to either the page, a String in 'var', or a Reader in 'varReader'.
<u><c.forEach></u>	The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality.
<u><c.forTokens></u>	Iterates over tokens, separated by the supplied delimiters.
<u><c.param></u>	Adds a parameter to a containing 'import' tag's URL.
<u><c.redirect></u>	Redirects to a new URL.
<u><c.url></u>	Creates a URL with optional query parameters

2. Formatting tags

The JSTL formatting tags are used to format and display text, the date, the time, and numbers for internationalized Web sites.

Following is the syntax to include formatting library in our JSP:

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

Tag	Description
<u><fmt.formatNumber></u>	To render numerical value with specific precision or format.
<u><fmt.parseNumber></u>	Parses the string representation of a number, currency, or percentage.
<u><fmt.formatDate></u>	Formats a date and/or time using the supplied styles and pattern
<u><fmt.parseDate></u>	Parses the string representation of a date and/or time
<u><fmt.bundle></u>	Loads a resource bundle to be used by its tag body.
<u><fmt.setLocale></u>	Stores the given locale in the locale configuration variable.
<u><fmt.setBundle></u>	Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable.

<u><fmt:timeZone></u>	Specifies the time zone for any time formatting or parsing actions nested in its body.
<u><fmt:setTimeZone></u>	Stores the given time zone in the time zone configuration variable
<u><fmt:message></u>	To display an internationalized message.
<u><fmt:requestEncoding></u>	Sets the request character encoding

3. SQL tags

The JSTL SQL tag library provides tags for interacting with relational databases (RDBMSs) such as Oracle, MySQL, or Microsoft SQL Server.

Following is the syntax to include JSTL SQL library in our JSP:

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

Tag	Description
<u><sql:setDataSource></u>	Creates a simple DataSource suitable only for prototyping
<u><sql:query></u>	Executes the SQL query defined in its body or through the sql attribute.
<u><sql:update></u>	Executes the SQL update defined in its body or through the sql attribute.
<u><sql:param></u>	Sets a parameter in an SQL statement to the specified value.
<u><sql:dateParam></u>	Sets a parameter in an SQL statement to the specified java.util.Date value.
<u><sql:transaction ></u>	Provides nested database action elements with a shared Connection, set up to execute all statements as one transaction.

4. XML tags

The JSTL XML tags provide a JSP-centric way of creating and manipulating XML documents. Following is the syntax to include JSTL XML library in our JSP. The JSTL XML tag library has custom tags for interacting with XML data. This includes parsing XML, transforming XML data, and flow control based on XPath expressions.

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

<u><x:out></u>	Like <%= ... >, but for XPath expressions.
<u><x:parse></u>	Use to parse XML data specified either via an attribute or in the tag body.
<u><x:set ></u>	Sets a variable to the value of an XPath expression.
<u><x:if ></u>	Evaluates a test XPath expression and if it is true, it processes its body. If the test condition is false, the body is ignored.
<u><x:forEach></u>	To loop over nodes in an XML document.
<u><x:choose></u>	Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>
<u><x:when ></u>	Subtag of <choose> that includes its body if its expression evaluates to 'true'
<u><x:otherwise ></u>	Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false'
<u><x:transform ></u>	Applies an XSL transformation on a XML document
<u><x:param ></u>	Use along with the transform tag to set a parameter in the XSLT stylesheet

5. JSTL Functions

JSTL includes a number of standard functions, most of which are common string manipulation functions.

Following is the syntax to include JSTL Functions library in our JSP:

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

Function	Description
<u>fn:contains()</u>	Tests if an input string contains the specified substring.
<u>fn:containsIgnoreCase()</u>	Tests if an input string contains the specified substring in a case insensitive way.
<u>fn:endsWith()</u>	Tests if an input string ends with the specified suffix.
<u>fn:escapeXml()</u>	Escapes characters that could be interpreted as XML markup.
<u>fn:indexOf()</u>	Returns the index withing a string of the first occurrence of a specified substring.
<u>fn:join()</u>	Joins all elements of an array into a string.
<u>fn:length()</u>	Returns the number of items in a collection, or the number of characters in a string.
<u>fn:replace()</u>	Returns a string resulting from replacing in an input string all occurrences with a given string.
<u>fn:split()</u>	Splits a string into an array of substrings.
<u>fn:startsWith()</u>	Tests if an input string starts with the specified prefix.
<u>fn:substring()</u>	Returns a subset of a string.
<u>fn:substringAfter()</u>	Returns a subset of a string following a specific substring.
<u>fn:substringBefore()</u>	Returns a subset of a string before a specific substring.
<u>fn:toLowerCase()</u>	Converts all of the characters of a string to lower case.
<u>fn:toUpperCase()</u>	Converts all of the characters of a string to upper case.
<u>fn:trim()</u>	Removes white spaces from both ends of a string.

10.6. Session Management in JSP

Session Handling becomes mandatory when a requested data need to be sustained for further use. Since http protocol considers every request as a new one, session handling becomes important.

The session object is used to track a client session between client requests.

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

Following are some of the methods to handle session:

- In JSP whenever a request arises the server generates a unique Session ID which is stored in the client machine.
- Cookies store the information in the client browser
- URL rewriting the session information is appended to the end of the URL
- Hidden form fields the sessionId is embedded to GET and POST command.

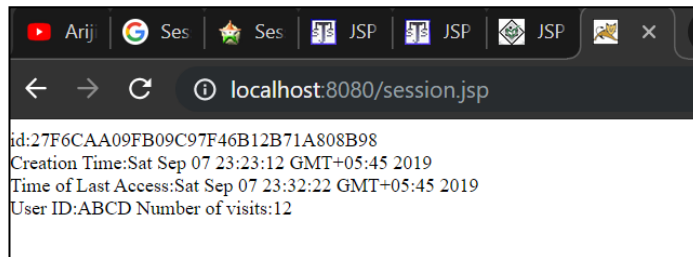
Save the below file as *session.jsp* at ROOT directory of *webapps* in Tomcat.

```
<%@ page import="java.io.*,java.util.*" %>
<%
    //Get session creation time and last Access time
    Date createTime = new Date(session.getCreationTime());
    Date lastAccessTime = new Date(session.getLastAccessedTime());

    String title = "Welcome Back";
    Integer visitCount = new Integer(0);
    String visitCountKey = "visitCount";
    String userIDKey = new String("userID");
    String userID = new String("Java2s_ID");

    // Check if this is new comer on your Webpage.
    if (session.isNew()){
        title = "Welcome";
        session.setAttribute(userIDKey, userID);
        session.setAttribute(visitCountKey, visitCount);
    }
    visitCount = (Integer)session.getAttribute(visitCountKey);
    visitCount = visitCount + 1;
    userID = (String)session.getAttribute(userIDKey);
    session.setAttribute(visitCountKey, visitCount);
%>
<html>
<body>
    id:<% out.print( session.getId()); %><br/>
    Creation Time:<% out.print(createTime); %><br/>
    Time of Last Access:<% out.print(lastAccessTime); %><br/>
    User ID:<% out.print(userID); %>
    Number of visits:<% out.print(visitCount); %><br/>
</body>
</html>
```

Output:



Assignment:

Differentiate between JSP and Servlet.

Reference: for more study:

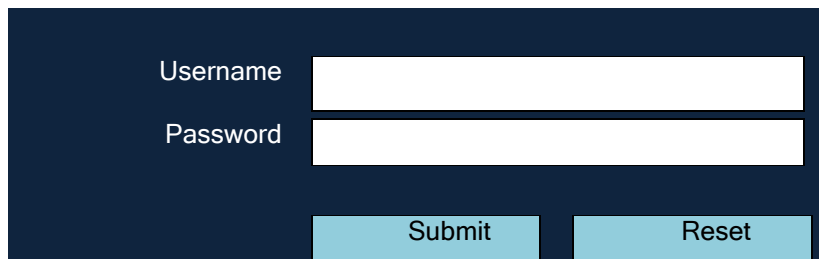
<https://www.youtube.com/watch?v=bd50C6XUnFw>

<https://www.wisdomjobs.com/e-university/adv-java-tutorial-227/the-javax-dot-servlet-dot-http-package-6187.html>

Exam Questions:

1. What is JSP tag library? Explain about session management. How many techniques are in session management? [2018 spring]
2. Explain JSP and servlet. What are the advantages of JSP? [2018 fall]

3. Define JavaServer Pages Standard Tag Library JSTL0 and Http Session Interface. Explain JSP Life Cycle. [2018 Fall]
4. Define JSP and Servlets. How are servlets created and deployed? Explain with an illustration. [2017 fall]
5. What are the session management techniques available in JSP? Explain briefly. [2017 spring]
6. Explain HTTP requests handling and session management in JSP. [2016 fall]
7. Explain about the life cycle of JSP with a neat diagram. [2016 fall]
8. Write a JSP application that submits the basic student registration form to another page called processRegister.jsp which should display all the records. [2016 fall]
9. Explain Servlet Life Cycle and How can you deploy Servlet in Java. [2016 spring]
10. Give the HTML to create a form with two elements: a textbox named FirstName that holds a maximum of 50 characters, a Submit button. The form should submit its data to a JSP program called Process.jsp using the POST method. Also, write the Process.jsp to handle these requests. [2016 spring, 2014 fall]
11. What are the uses of session? How is it managed in JSP? Explain with example. [2015 spring]
12. Write a code for following example using Servlet:



Username

Password

13. What are uses of sessions? How is it managed in JSP? Explain with examples. [2015 fall]
14. What is JSP and Servlet technology? Explain in brief about the advantages of Servlet over the CGI technology. [2014 spring]
15. What are the interfaces of HTTP request handling? Mention each with their name and short description. Explain in brief about servlet life cycle. [2014 spring]
16. How does JSP differ with Java Servlet? Explain the life cycle of JSP. [2014 fall]
17. Explain JSP core tags, formatting tags, XML tags and SQL tags with suitable examples. Illustrate. [2013 spring]
18. Define session and how do we manage session in Servlet? [2013 spring]
19. Write short notes on:
 - a. JSP syntax [2018 spring]
 - b. Session Management [2017 fall]
 - c. Servlet Life Cycle [2017, 2015 spring]
 - d. JSP [2015 spring]
