ROHINI COLLEGE OF ENGINEERING & TECHNOLOGY
Near Anjugramam Junction, Kanyakumari Main Road, Palkulam, Variyoor P.O - 629401
Kanyakumari Dist, Tamilnadu., E-mail : admin@rcet.org.in, Website : www.rcet.org.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NAME OF THE SUBJECT : Internet Programming

Subject code : CS8651

Regulation : 2017

UNIT III SERVER SIDE PROGRAMMING

UNIT-3

SERVER SIDE PROGRAMMING

JAVA SERVLET ARCHITECTURE AND ITS LIFE CYCLE

- 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.Java Servlets often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But Servlets offer several advantages in comparison with the CGI.
- 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.
- Servlets are platform-independent because they are written in Java.
- Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. So servlets are trusted.
- 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 mechanisms that you have seen already.

Servlets architecture

Following diagram shows the position of Servelts in a Web Application.

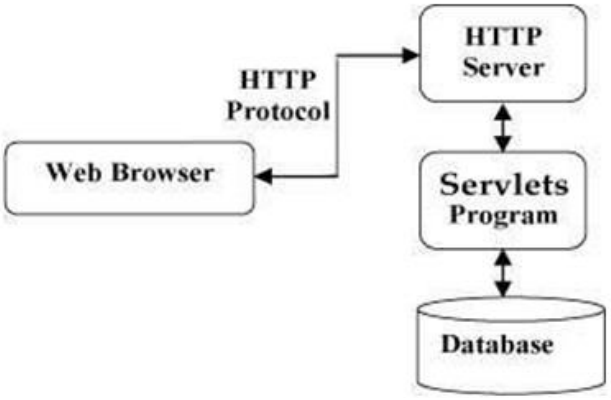
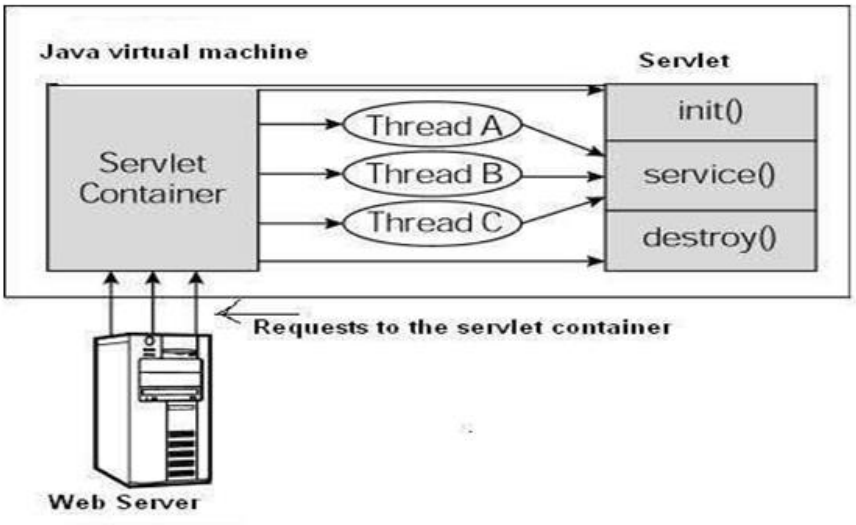


Fig: servlet life-cycle scenario

- 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.

Servlets tasks

Servlets perform the following major tasks:

- Read the explicit data sent by the clients (browsers). This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.
- Read the implicit HTTP request data sent by the clients (browsers). This includes cookies, media types and compression schemes the browser understands, and so forth.
- Process the data and generate the results. This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.
- Send the explicit data (i.e., the document) to the clients (browsers). This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), Excel, etc.
- Send the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Servlets packages

- Java Servlets are Java classes run by a web server that has an interpreter that supports the Java Servlet specification.
- Servlets can be created using the **javax.servlet** and **javax.servlet.http** packages, which are a standard part of the Java's enterprise edition, an expanded version of the Java class library that supports large-scale development projects.
- These classes implement the Java Servlet and JSP specifications. At the time of writing this tutorial, the versions are Java Servlet 2.5 and JSP 2.1.
- Java servlets have been created and compiled just like any other Java class. After you install the servlet packages and add them to our computer's Classpath, we can compile servlets with the JDK's Java compiler or any other current compiler.

Life cycle of the servlet

The web container maintains the life cycle of a servlet instance.

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.

As displayed in the diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

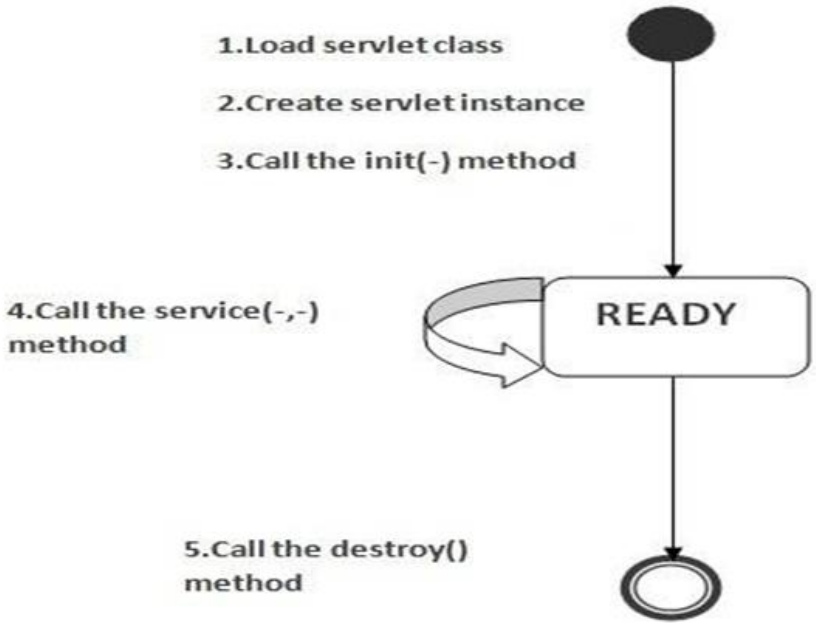
2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is

given below:
public void init(ServletConfig config) **throws** ServletException



4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

public void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException

5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

public void destroy()

COOKIES IN SERVLET

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.

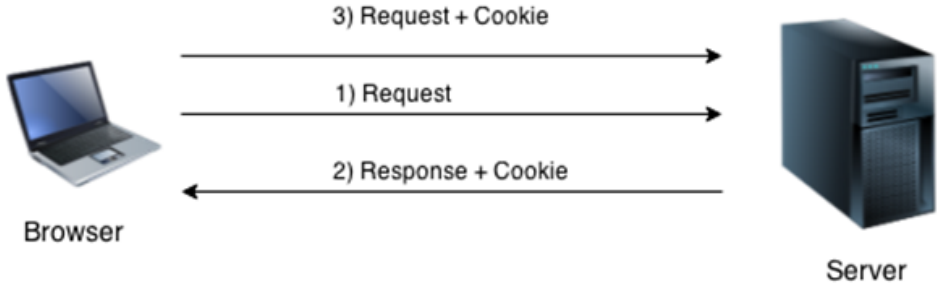
How Cookie works

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.

Types of Cookie

There are 2 types of cookies in servlets.

1. Non-persistent cookie
2. Persistent cookie



Non-persistent cookie
It is **valid for single session** only. It is removed each time when user closes the browser.

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

Disadvantage of Cookies
1. It will not work if cookie is disabled from the browser.
2. Only textual information can be set in Cookie object.

Cookie class
javax.servlet.http.Cookie class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

Constructor of Cookie class

Constructor	Description
Cookie()	constructs a cookie.
Cookie(String name, String value)	constructs a cookie with a specified name and value.

Useful Methods of Cookie class
There are given some commonly used methods of the Cookie class.

Method	Description
public void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.
public String getName()	Returns the name of the cookie. The name cannot be changed after creation.
public String getValue()	Returns the value of the cookie.
public void setName(String name)	changes the name of the cookie.
public void setValue(String value)	changes the value of the cookie.

Other methods required for using Cookies
For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces. They are:
1. **public void addCookie(Cookie ck):**method of HttpServletResponse interface is used to add cookie in response object.
2. **public Cookie[] getCookies():**method of HttpServletRequest interface is used to return all the cookies from the browser.

Creation of Cookie
Let's see the simple code to create cookie.

```
1. Cookie ck=new Cookie("user","sonoo jaiswal");          //creating cookie object
2. response.addCookie(ck);                                //adding cookie in the response
```

Deletion of Cookie

Let's see the simple code to delete cookie. It is mainly used to logout or signout the user.

```
1. Cookie ck=new Cookie("user","");          //deleting value of cookie
2. ck.setMaxAge(0);                          //changing the maximum age to 0 seconds
3. response.addCookie(ck);                   //adding cookie in the response
```

Example of Cookies in java servlet

index.html

```
<form action="login">
 User Name:<input type="text" name="userName"/><br/>
 Password:<input type="password" name="userPassword"/><br/>
 <input type="submit" value="submit"/>
</form>
```

MyServlet1.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet1 extends HttpServlet
{
  public void doGet(HttpServletRequest request,
    HttpServletResponse response) {
    try{
        response.setContentType("text/html");
        PrintWriter pwriter = response.getWriter();

        String name = request.getParameter("userName");
        String password = request.getParameter("userPassword");
        pwriter.print("Hello "+name);
        pwriter.print("Your Password is: "+password);

        //Creating two cookies
        Cookie c1=new Cookie("userName",name);
        Cookie c2=new Cookie("userPassword",password);

        //Adding the cookies to response header
        response.addCookie(c1);
        response.addCookie(c2);
        pwriter.print("<br><a href='welcome'>View Details</a>");
        pwriter.close();
  }catch(Exception exp){
     System.out.println(exp);
   }
 }
}
```

MyServlet2.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet2 extends HttpServlet {
 public void doGet(HttpServletRequest request,
   HttpServletResponse response){
   try{
      response.setContentType("text/html");
      PrintWriter pwriter = response.getWriter();

      //Reading cookies
      Cookie c[]=request.getCookies();
      //Displaying User name value from cookie
      pwriter.print("Name: "+c[1].getValue());
      //Displaying user password value from cookie
      pwriter.print("Password: "+c[2].getValue());

      pwriter.close();
   }catch(Exception exp){
      System.out.println(exp);
    }
  }
}
```

web.xml

```
<web-app>
<display-name>BeginnersBookDemo</display-name>
 <welcome-file-list>
 <welcome-file>index.html</welcome-file>
 </welcome-file-list>
<servlet>
 <servlet-name>Servlet1</servlet-name>
 <servlet-class>MyServlet1</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Servlet1</servlet-name>
 <url-pattern>/login</url-pattern>
</servlet-mapping>
<servlet>
 <servlet-name>Servlet2</servlet-name>
 <servlet-class>MyServlet2</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>Servlet2</servlet-name>
 <url-pattern>/welcome</url-pattern>
</servlet-mapping>
</web-app>
```

Output:
Welcome Screen:

After clicking Submit:

After clicking View Details:

SESSION

A session is a conversation between the server and a client. A conversation consists series of continuous request and response.

Why should a session be maintained?

When there is a series of continuous request and response from a same client to a server, the server cannot identify from which client it is getting requests. Because HTTP is a stateless protocol.

When there is a need to maintain the conversational state, session tracking is needed. For example, in a shopping cart application a client keeps on adding items into his cart using multiple requests. When every request is made, the server should identify in which client's cart the item is to be added. So in this scenario, there is a certain need for session tracking.

Solution is, when a client makes a request it should introduce itself by providing unique identifier every time. There are five different methods to achieve this.

Session tracking methods:

1. User authorization
2. Hidden fields
3. URL rewriting

4. Cookies
5. Session tracking API

The first four methods are traditionally used for session tracking in all the server-side technologies. The session tracking API method is provided by the underlying technology (java servlet or PHP or likewise). Session tracking API is built on top of the first four methods.

1. User Authorization

Users can be authorized to use the web application in different ways. Basic concept is that the user will provide username and password to login to the application. Based on that the user can be identified and the session can be maintained.

2. Hidden Fields

<INPUT TYPE="hidden" NAME="technology" VALUE="servlet">
Hidden fields like the above can be inserted in the webpages and information can be sent to the server for session tracking. These fields are not visible directly to the user, but can be viewed using view source option from the browsers. This type doesn't need any special configuration from the browser of server and by default available to use for session tracking. This cannot be used for session tracking when the conversation included static resources lik html pages.

3. URL Rewriting

Original URL: http://server:port/servlet/ServletName
Rewritten URL: http://server:port/servlet/ServletName?sessionid=7456
When a request is made, additional parameter is appended with the url. In general added additional parameter will be sessionid or sometimes the userid. It will suffice to track the session. This type of session tracking doesn't need any special support from the browser. Disadvantage is, implementing this type of session tracking is tedious. We need to keep track of the parameter as a chain link until the conversation completes and also should make sure that, the parameter doesn't clash with other application parameters.

4. Cookies

Cookies are the mostly used technology for session tracking. Cookie is a key value pair of information, sent by the server to the browser. This should be saved by the browser in its space in the client computer. Whenever the browser sends a request to that server it sends the cookie along with it. Then the server can identify the client using the cookie.
In java, following is the source code snippet to create a cookie:

Cookie cookie = new Cookie("userID", "7456"); res.addCookie(cookie);

Session tracking is easy to implement and maintain using the cookies. Disadvantage is that, the users can opt to disable cookies using their browser preferences. In such case, the browser will not save the cookie at client computer and session tracking fails.

5. Session tracking API

Session tracking API is built on top of the first four methods. This is inorder to help the developer to minimize the overhead of session tracking. This type of session tracking is provided by the underlying technology. Lets take the java servlet example. Then, the servlet container manages the session tracking task and the user need not do it explicitly using the java servlets.

This is the best of all methods, because all the management and errors related to session tracking will be taken care of by the container itself.

Every client of the server will be mapped with a javax.servlet.http.HttpSession object. Java servlets can use the session object to store and retrieve java objects across the session. Session tracking is at the best when it is implemented using session tracking api.

```
package com.journaldev.servlet.session;
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class LoginServlet
 */ @WebServlet("/LoginServlet")
public class LoginServlet extends HttpServlet {
 private static final long serialVersionUID = 1L;
private final String userID = "Pankaj";
private final String password = "journaldev";
protected void doPost(HttpServletRequest request,HttpServletResponse response) throws
   ServletException, IOException {

// get request parameters for userID and password
String user = request.getParameter("user");
String pwd = request.getParameter("pwd");
      if(userID.equals(user) && password.equals(pwd)){
      Cookie loginCookie = new Cookie("user",user);
         //setting cookie to expiry in 30 mins loginCookie.setMaxAge(30*60);
         response.addCookie(loginCookie);
         response.sendRedirect("LoginSuccess.jsp");
      }else
      {
         RequestDispatcher rd = getServletContext().getRequestDispatcher("/login.html");
         PrintWriter out= response.getWriter();
         out.println("<font color=red>Either user name or password is wrong.</font>");
         rd.include(request, response);
      }

   }

}
```

SERVLETS - FORM DATA

We can read html ***form*** data from a URL and process it in a servlet and then send the response back to the client.

GET and POST method

As seen earlier in Servlet terminologies, GET and POST methods are used to pass information from a form to a Java Servlet. While using GET method the *form data is passed in the url as query parameters*. GET method is the *default* method used. It looks like,

http://localhost:8080/hello?key1=value1&key2=value2

If POST method is used then the *form data is passed in the HTTP request message body*. It cannot be seen in the URL as seen in GET method. URL length has a limitation and so if we are passing large volume of data we should use POST. Similarly *sensitive data like passwords should also be passed using POST* method.

Read Form Data

we have methods doXYZ(…) in the HTTPServlet class. The method doGet(…) is invoked by the servlet container if the form uses GET method or doPost(…) if the the form uses POST method. In this example we are using GET method.

```
package com.javapapers.servlet.introduction;
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;

public class HelloFormData extends HttpServlet {
	private static final long serialVersionUID = 1L;
	public void doGet(HttpServletRequest request, HttpServletResponse response)
				throws ServletException, IOException
	{
		response.setContentType("text/html");
		PrintWriter out = response.getWriter();
		String title = "Servlet: Read Form Data";
		out.println("<html>" + "<head><title>" + title
					+ "</title></head><body>"
					+ "<h1>" + title + "</h1>"
					+ "<p>Hi "
					+ request.getParameter("name")
					+ "</p>"
					+ "</body></html>");
	}
}
```

web.xml

This web.xml defines the Servlet mapping for the form data servlet.

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
     http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
```

```
    <display-name>Servlet Form Data Handling</display-name>

    <servlet>
        <servlet-name>HelloFormData</servlet-name>
        <servlet-class>com.javapapers.servlet.introduction.HelloFormData</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloFormData</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```

index.html

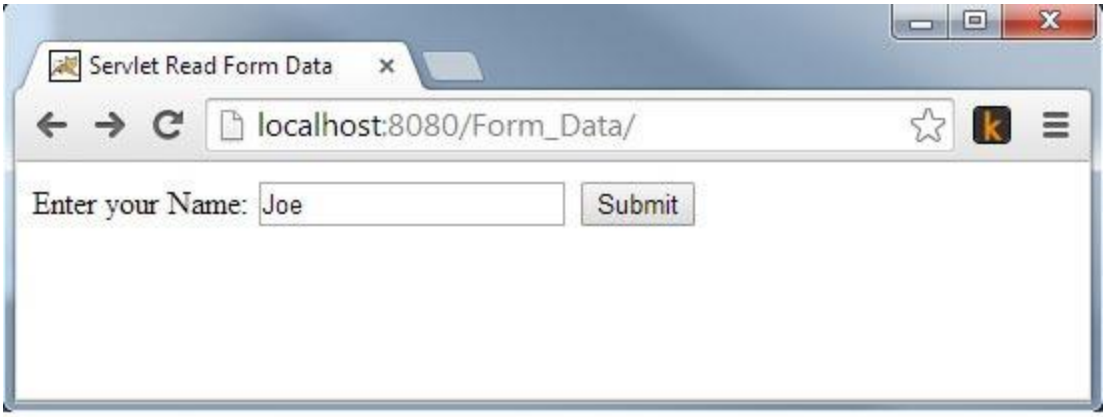
```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Servlet Read Form Data</title>
</head>
<body>
<form action="./hello" method="GET">
Enter your Name: <input type="text" name="name">
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

Read using POST method

In the above HTML page in form tag instead of GET use as POST. Then in the servlet to read the form data add a doPost method as below,

```
package com.javapapers.servlet.introduction;
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;

public class HelloFormData extends HttpServlet
{
        private static final long serialVersionUID = 1L;
        public void doGet(HttpServletRequest request, HttpServletResponse response)
                        throws ServletException, IOException
        {
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
                String title = "Servlet: Read Form Data";
                out.println("<html>" + "<head><title>" + title
                                + "</title></head><body>"
                                + "<h1>" + title + "</h1>"
                                + "<p>Hi "
                                + request.getParameter("name")
                                + "</p>"
                                + "</body></html>");
        }
```

```
public void doPost(HttpServletRequest request,HttpServletResponse response)throws ServletException, IOException
{
    doGet(request, response);
 }
}
```



INSTALL AND CONFIGURING APACHE TOMCAT WEB SERVER

Apache Tomcat is a Java-capable HTTP server, which is able to execute special Java programs known as Java Servlet and Java Server Pages (JSP). It is also able to execute HTML files just like Apache HTTP.

Requirements

To illustrate the steps below a Windows 7 64-bit computer was used. The Windows computer had Internet access. Tomcat 8 was used for this installation.

We need to have the latest Java JDK version installed. The recommended JDK version for Tomcat 8 is jdk1.8. Make sure that your computer is updated with this version. Go to java.com to download the latest version of Java.

The steps below are for a fresh installation of Apache Tomcat

Tomcat Setup

i. Download and Install Tomcat

Go to http://tomcat.apache.org/ and download the latest version of Tomcat (Tomcat 9 at the time of writing this). Under Download on the left click on 'Tomcat 9' and under Binary Distribution click on 'Core' and you will see various packages. Click on the package applicable to you. For a Windows 64-bit computer click on '64- bit Windows zip'. Download this file.

Unzip the downloaded file to a directory of your choice. It is recommended not to unzip to the desktop as it is a difficult directory to locate from a command prompt.

For this illustration a folder named ‘tomcat’ was created under drive D, hence D:/tomcat. The zipped file was extracted to this location. It is recommended to rename the unzipped file from the default name, for example, rename to ‘apachetomcat’ (Figure 10).

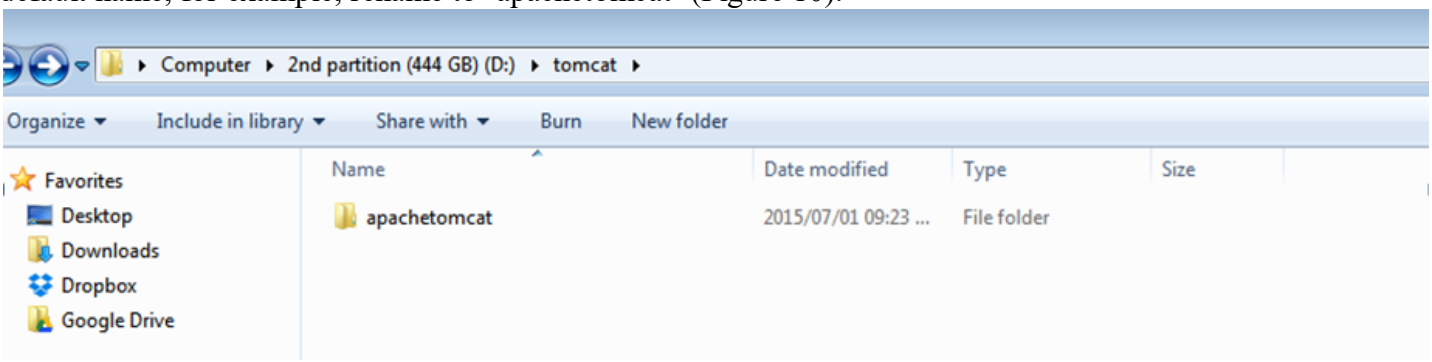


Figure 10: Creating folders to store tomcat files

ii. Create an Environment Variable

First, take note of the directory into which JDK was installed. The default is "C:\Program Files\Java\jdk1.8.0_{xx}", where {xx} is the latest upgrade number. It is important to verify your JDK installed directory before you proceed further. Start the command prompt and type ‘set JAVA_HOME to check if the environmental variable had been set. If not, you will get the message ‘Environment Variable JAVA_HOME not set’. If JAVA_HOME is set, check if it is set to your JDK installed directory correctly (For example in Figure 11). If not, proceed to set the environmental variable as below.

To set the environment variable JAVA_HOME in Windows 7: Press "Start" button > Control Panel > System & Security > System > Advanced system settings > Switch to "Advanced" tab > Environment Variables > System Variables > "New" (or "Edit" for modification) > In "Variable Name", enter "JAVA_HOME" > In "Variable Value", enter your JDK installed directory (e.g., "c:\Program Files\Java\jdk1.8.0_51").

Click OK. To verify, **restart** the command prompt and type ‘set JAVA_HOME’. You should now see the output as in Figure 11.

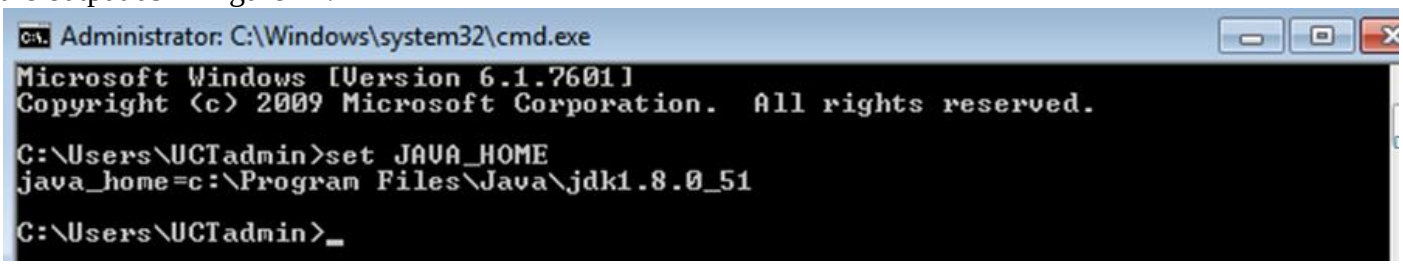


Figure 11: Checking environmental variables for JAVA_HOME

iii. Configure Tomcat Server

The Tomcat configuration files are located in the "conf" sub-directory of your Tomcat installed directory, e.g. "D:\tomcat\apachetomcat\conf". There are 4 configuration XML files: server.xml, web.xml, context.xml and tomcat-users.xml. Make a BACKUP of the configuration files before you proceed.

Set the TCP Port Number

Use an HTML text editor (e.g., NotePad++) to open the configuration file "server.xml", under the "conf" sub- directory of Tomcat installed directory.

The default TCP port number configured in Tomcat is 8080, you may choose any number between 1024 and 65535, which is not used by an existing application. We shall choose 8888 in this article. Locate the following lines that define the HTTP connector, and change port="8080" to port="8888". Save the file and exit.

<Connector port="8888" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443" />

Enabling Directory Listing

Again, use an HTML text editor to open the configuration file "web.xml", under the "conf" sub-directory of Tomcat installed directory.

We shall enable directory listing by changing "listings" from "false" to "true" for the "default" servlet. Locate the following lines that define the "default" servlet; and change the "listings" from "false" to "true". Save and exit.

<servlet>

<servlet-name>default</servlet-name>

<servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
<init-param>
<param-name>debug</param-name>

<param-value>0</param-value>

</init-param>
<init-param>
<param-name>listings</param-name>

<param-value>true</param-value>

</init-param>
<load-on-startup>1</load-on-startup>
</servlet>

Enabling Automatic Reload

We shall add the attribute reloadable="true" to the <Context> element to enable automatic reload after code changes. Again, this is handy for test system but not for production, due to the overhead of detecting changes. Open context.xml and locate the <Context> start element, and change it to <Context reloadable="true">. Save and exit.

<Context reloadable = "true">

…… </Context>

iv. Start Tomcat Server

Launch a CMD shell. Set the current directory to the tomcat directory\bin", and run "startup.bat" as in Figure 12:

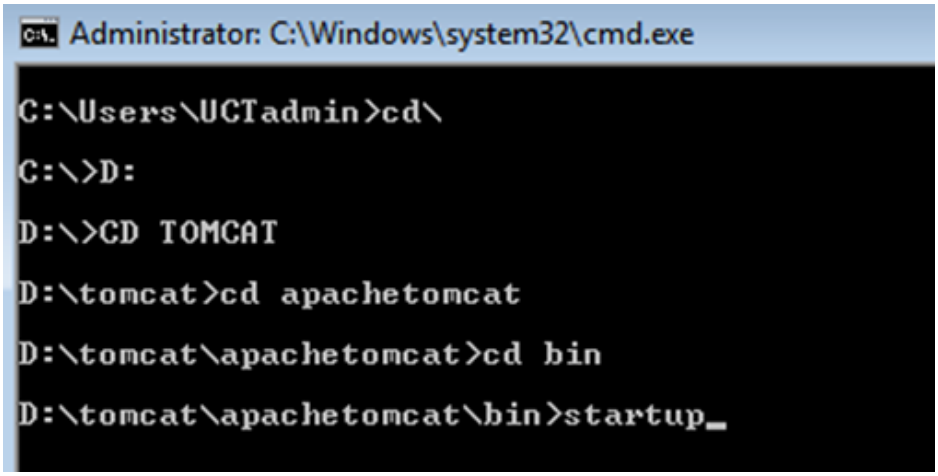


Figure 12: Starting tomcat from command prompt

A new Tomcat console window appears (look out for the Tomcat's port number (double check that Tomcat is running on port 8888). Future error messages will be sent to this console. Output messages from related Java programs are also sent to this console. If you want to shut down the server type 'shutdown' in place of 'startup'.

Start a browser. Issue URL "http://localhost:8888" to access the Tomcat server's welcome page. For users on the other machines over the net, they have to use the server's IP address or DNS domain name or hostname in the format of "http://serverHostnameOrIPAddress:8888". Note that this may not work if you are sharing the address with someone outside of your network who is behind a firewall. If everything is setup correctly, you should see the screen in Figure 13.

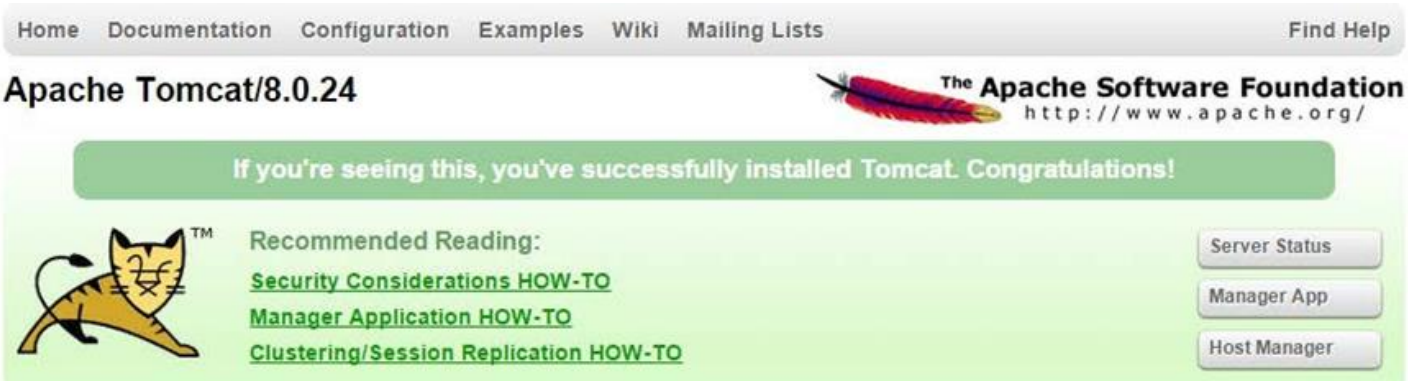


Figure 13: Tomcat home screen on local host

Install Tomcat's Sample Web Application

Go to: http://localhost:8888/docs/

Click the link: "3. First web application"

Click "Example App" under contents on the left side of the screen.

Click on the link "here" to download their "Sample Application".

(Download link: http://localhost:8888/docs/appdev/sample/sample.war)

Save to this location: C:\Tomcat\apachetomcat\webapps
Give the Tomcat container a minute and it will automatically extract the WAR file and create a Web Application called "Sample".

Test your install: http://localhost:8888/sample

If you wish to create your own directory under webapps, choose a *name* for your webapp. Let's call it "myapps". Go to Tomcat's "webapps" sub-directory. Create the following directory structure for you webapp "myapps":

1. Under Tomcat's "webapps", create your webapp *root* directory"myapps" (i.e., "tomcat\apachetomcat\webapps\myapps").
2. Under "myapps", create a sub-directory "WEB-INF" (case sensitive, a "dash" not an underscore) (i.e., "tomcat\apachetomcat\webapps\myapps\WEB-INF").
3. Under "WEB-INF", create a sub-sub-directory "classes" (case sensitive, plural) (i.e., "tomcat\apachetomcat\webapps\myapps\WEB-INF\classes").

Restart your tomcat server to pick up the changes. Then on your browser type http://localhost:8888/myapps/
You should see the directory listing of the directory "tomcat\apachetomcat\webapps\myapps", which shall be empty

---

DATABASE CONNECTIVITY

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

Structured query language using MySQL:

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Add a database called books, enter:
mysql> **CREATE DATABASE books;**

Now, database is created. Use a database with use command, type:
mysql> **USE books;**

Next, create a table called authors with name, email and id as fields:
mysql> **CREATE TABLE authors (id INT, name VARCHAR(20), email VARCHAR(20));**

To display your tables in books database, enter:
mysql> **SHOW TABLES;**
Sample outputs:

```
+-----------------+
| Tables_in_books |
+-----------------+
| authors         |
+-----------------+
1 row in set (0.00 sec)
```

To display various fields in author table
mysql> **DESCRIBE author;**

```
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| id      | INT         | YES  |     | NULL    |       |
| name    | varchar(20) | YES  |     | NULL    |       |
| email   | varchar(20) | YES  |     | NULL    |       |
| +---------+-------------+------+-----+---------+----+
```

Finally, add a data i.e. row to table books using INSERT statement, run:
mysql> **INSERT INTO authors (id,name,email) VALUES(1,"Vivek","xuz@abc.com");**
Sample outputs:

```
Query OK, 1 row affected (0.00 sec)
```

Try to add few more rows to your table:
mysql> **INSERT INTO authors (id,name,email) VALUES(2,"Priya","p@gmail.com");**
mysql> **INSERT INTO authors (id,name,email) VALUES(3,"Tom","tom@yahoo.com");**

To display all rows i.e. data stored in authors table, enter:
mysql> **SELECT * FROM authors;**
Sample outputs:

```
+------+-------+---------------+
| id   | name  | email         |
+------+-------+---------------+
|    1 | Vivek | xuz@abc.com   |
|    2 | Priya | p@gmail.com   |
|    3 | Tom   | tom@yahoo.com |
+------+-------+---------------+
3 rows in set (0.00 sec)
```

To delete a particular row
mysql>DELETE FROM author WHERE ID=3;
mysql>SELECT *from author;

```
+------+-------+---------------+
| id   | name  | email         |
+------+-------+---------------+
|    1 | Vivek | xuz@abc.com   |
|    2 | Priya | p@gmail.com   |
+------+-------+---------------+
2 rows in set (0.00 sec)
```

To delete a table author
mysql>drop table authors;

JDBC perspective:

Fundamentally, JDBC is a specification that provides a *complete set of interfaces* that allows for portable access to an underlying database. Java can be used to write different types of executables, such as

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database- independent code.

JDBC Architecture

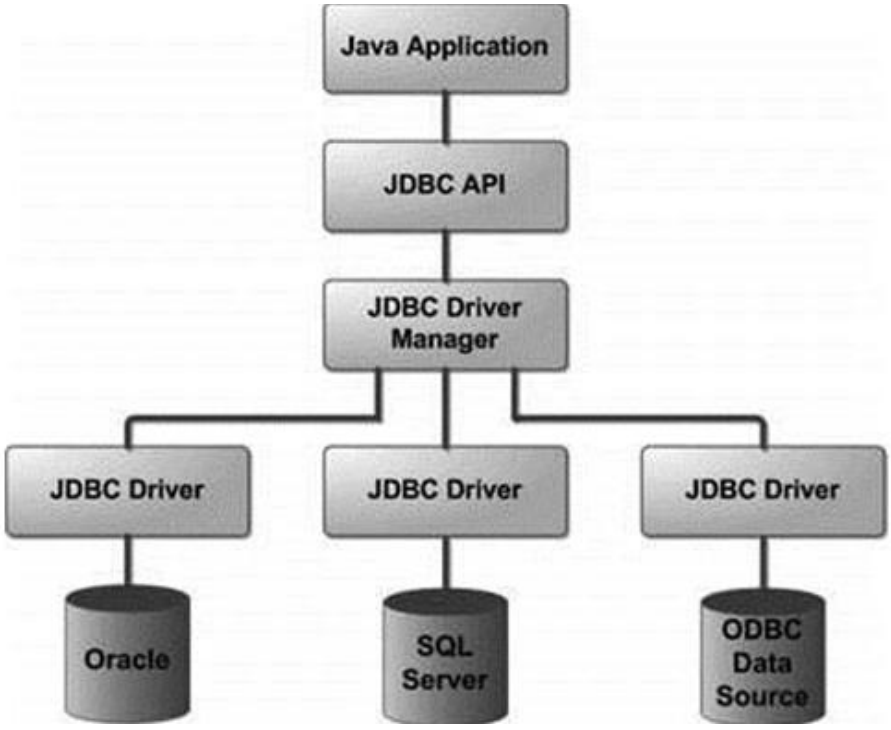
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application



Common JDBC Components

The JDBC API provides the following interfaces and classes −

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

Steps in JDBC

1. Import JDBC packages.
2. Load and register the JDBC driver.
3. Open a connection to the database.
4. Create a statement object to perform a query.
5. Execute the statement object and return a query resultset.
6. Process the resultset.
7. Close the resultset and statement objects.
8. Close the connection.

Java Server Pages(JSP)

- It stands for **Java Server Pages**.
- It is a server side technology.
- It is used for creating web application.
- It is used to create dynamic web content.
- In this JSP tags are used to insert JAVA code into HTML pages.
- It is an advanced version of Servlet Technology.
- It is a Web based technology helps us to create dynamic and platform independent web pages.
- In this, Java code can be inserted in HTML/ XML pages or both.
- JSP is first converted into servlet by JSP container before processing the client's request.

JSP pages are more advantageous than Servlet:

- They are easy to maintain.
- No recompilation or redeployment is required.
- JSP has access to entire API of JAVA .
- JSP are extended version of Servlet.

Features of JSP

- **Coding in JSP is easy** :- As it is just adding JAVA code to HTML/XML.
- **Reduction in the length of Code** :- In JSP we use action tags, custom tags etc.
- **Connection to Database is easier** :-It is easier to connect website to database and allows to read or write data easily to the database.
- **Make Interactive websites :-** In this we can create dynamic web pages which helps user to interact in real time environment.
- **Portable, Powerful, flexible and easy to maintain** :- as these are browser and server independent.
- **No Redeployment and No Re-Compilation :-** It is dynamic, secure and platform independent so no need to re-compilation.
- **Extension to Servlet** :- as it has all features of servlets, implicit objects and custom tags

JSP syntax

Scripts which are present inside <% ... %> are called *scriptlet*. Syntax available in JSP are following

1. **Declaration Tag** :-It is used to declare variables.

```
Syntax:-
<%!  Dec var  %>
Example:-
<%! int var=10; %>
```

2. **Java Scriplets** :- It allows us to add any number of JAVA code, variables and expressions.

```
 Syntax:-
<% java code %>
```

3. **JSP Expression** :- It evaluates and convert the expression to a string.

```
Syntax:-
<%= expression %>
 Example:-
<% num1 = num1+num2 %>
```

4. **JAVA Comments** :- It contains the text that is added for information which has to be ignored.

```
 Syntax:-
<% -- JSP Comments %>
```

Process of Execution

Steps for Execution of JSP are following:-

- Create html page from where request will be sent to server eg try.html.
- To handle to request of user next is to create .jsp file Eg. new.jsp
- Create project folder structure.
- Create XML file eg my.xml.
- Create WAR file.
- Start Tomcat

- Run Application

Example

```
<%@page language="java" contentType="text/html"%>
<%! int day = 3; %>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <% if (day == 1 || day == 7) { %>
      <p> Today is weekend</p>
    <% } else { %>
      <p> Today is not weekend</p>
    <% } %>
  </body>
</html>
```

Advantages of using JSP

- It does not require advanced knowledge of JAVA
- It is capable of handling exceptions
- Easy to use and learn
- It can tags which are easy to use and understand
- Implicit objects are there which reduces the length of code
- It is suitable for both JAVA and non JAVA programmer

Disadvantages of using JSP

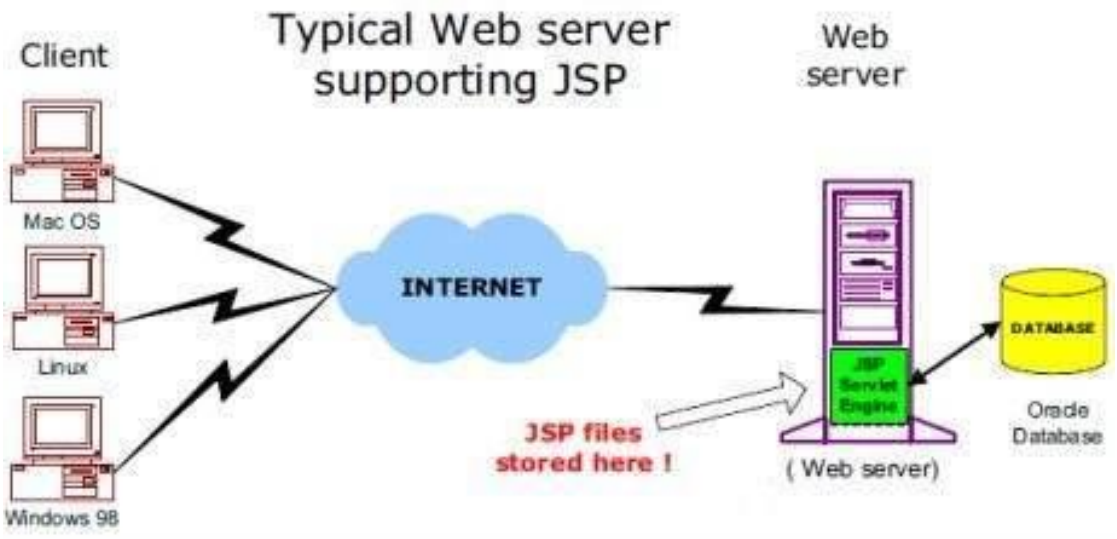
- Difficult to debug for errors.
- First time access leads to wastage of time
- It's output is HTML which lacks features.

Jsp Architecture:

The web server needs a JSP engine, i.e, a container to process JSP pages. The JSP container is responsible for intercepting requests for JSP pages. This tutorial makes use of Apache which has built-in JSP container to support JSP pages development.

A JSP container works with the Web server to provide the runtime environment and other services a JSP needs. It knows how to understand the special elements that are part of JSPs.

Following diagram shows the position of JSP container and JSP files in a Web application.

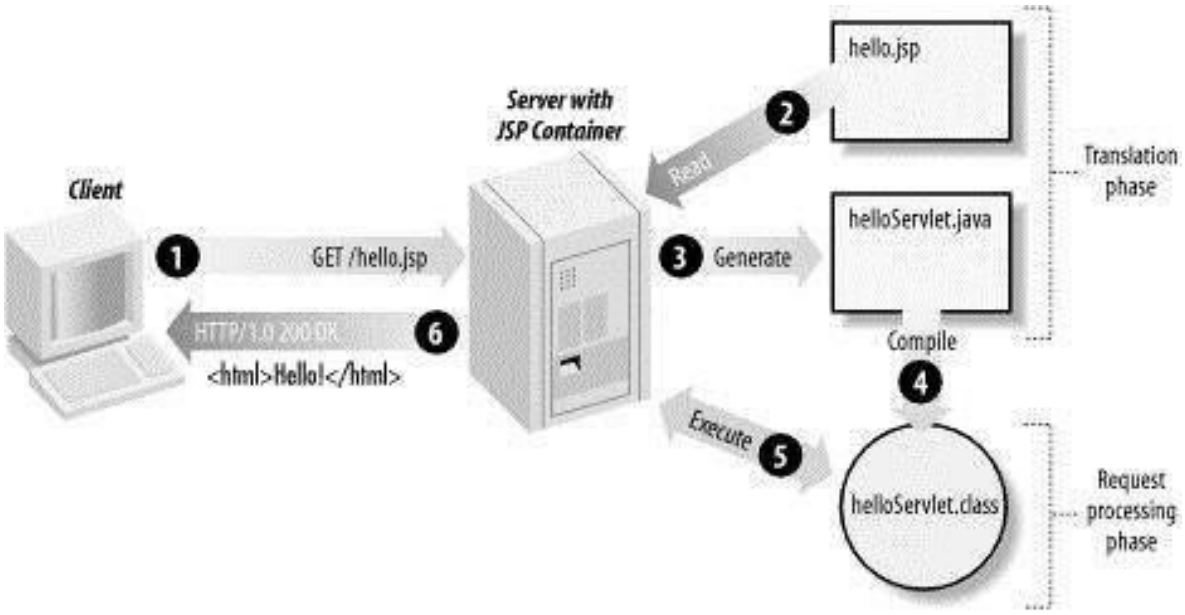


JSP Processing

The following steps explain how the web server creates the Webpage using JSP −

- As with a normal page, your browser sends an HTTP request to the web server.
- The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with **.jsp** instead of **.html**.
- The JSP engine loads the JSP page from disk and converts it into a servlet content. This conversion is very simple in which all template text is converted to println( ) statements and all JSP elements are converted to Java code. This code implements the corresponding dynamic behavior of the page.
- The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.
- A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format. The output is furthur passed on to the web server by the servlet engine inside an HTTP response.
- The web server forwards the HTTP response to your browser in terms of static HTML content.
- Finally, the web browser handles the dynamically-generated HTML page inside the HTTP response exactly as if it were a static page.

All the above mentioned steps can be seen in the following diagram −



Typically, the JSP engine checks to see whether a servlet for a JSP file already exists and whether the modification date on the JSP is older than the servlet. If the JSP is older than its generated servlet, the JSP container assumes that the JSP hasn't changed and that the generated servlet still matches the JSP's contents. This makes the process more efficient than with the other scripting languages (such as PHP) and therefore faster.

So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming wiz. Except for the translation phase, a JSP page is handled exactly like a regular servlet.

JSP Life Cycle:

The key to understanding the low-level functionality of JSP is to understand the simple life cycle they follow.

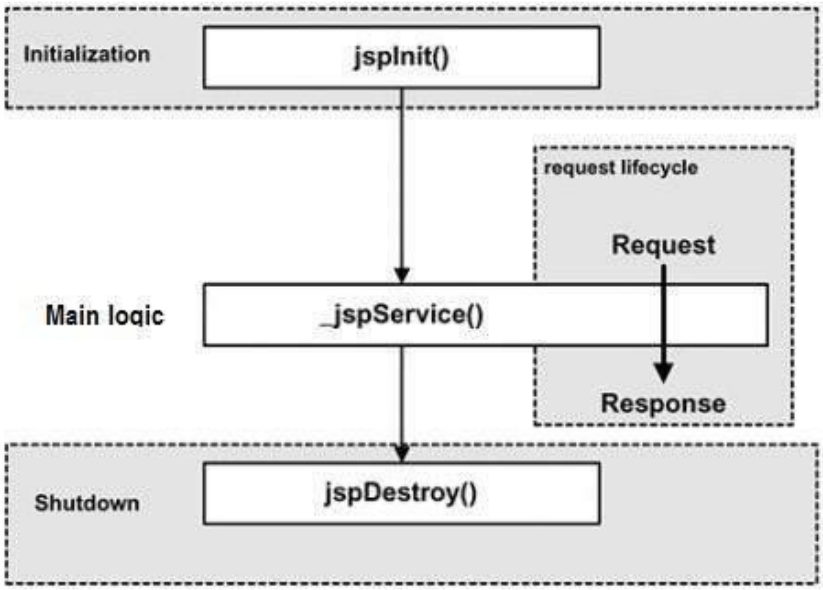
A JSP life cycle is defined as the process from its creation till the destruction. This is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet.

Paths Followed By JSP

The following are the paths followed by a JSP

- Compilation
- Initialization
- Execution
- Cleanup

The four major phases of a JSP life cycle are very similar to the Servlet Life Cycle. The four phases have been described below



JSP Compilation

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

The compilation process involves three steps

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

JSP Initialization

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

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

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

JSP Execution

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the **_jspService()** method in the JSP.

The _jspService() method takes an **HttpServletRequest** and an **HttpServletResponse** as its parameters as follows −

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

The **_jspService()** method of a JSP is invoked on request basis. This is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods, i.e, **GET, POST, DELETE**, etc.

JSP Cleanup

The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.

The **jspDestroy()** method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.

The jspDestroy() method has the following form −

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

JSP Directives

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

<%@ directive attribute="value" %>
There are three types of directive tag −

S.No.	Directive & Description
1	**<%@ page ... %>** Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
2	**<%@ include ... %>** Includes a file during the translation phase.
3	**<%@ taglib ... %>** Declares a tag library, containing custom actions, used in the page

JSP - The page Directive

The **page** directive is used to provide instructions to the container. These instructions pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.

Following is the basic syntax of the page directive −

<%@ page attribute = "value" %>

Example:

```
<%@page language=”java” contentType=”text/html”%>
<%! int day = 3; %>
<html>
  <head>
    <title>IF...ELSE Example</title>
  </head>
  <body>
    <% if (day == 1 || day == 7) { %>
      <p> Today is weekend</p>
    <% } else { %>
      <p> Today is not weekend</p>
    <% } %>
  </body>
</html>
```

The include Directive

The **include** directive is used to include a file during the translation phase. This directive tells the container to merge the content of other external files with the current JSP during the translation phase. You may code the ***include*** directives anywhere in your JSP page.

The general usage form of this directive is as follows −

<%@ include file = "relative url" >

main.jsp

```
<center>
  <p>Thanks for visiting my page.</p>
</center>
<%@ include file = "footer.jsp" %>
```

footer.jsp

```
    <br/><br/>
    <center>
      <p>Copyright © 2010</p>
    </center>
  </body>
</html>
```

Output:

Thanks for visiting my page.

Copyright © 2010

The taglib Directive

The JavaServer Pages API allow you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior.

The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides means for identifying the custom tags in your JSP page.

The taglib directive follows the syntax given below

```
<%@ taglib uri="uri" prefix = "prefixOfTag" >
```

Example:

```
<%@ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c" %>
<html>
  <head>
    <title><c:if> Tag Example</title>
  </head>
  <body>
    <c:set var = "salary" scope = "session" value = "${2000*2}"/>
    <c:if test = "${salary > 2000}">
      <p>My salary is:  <c:out value = "${salary}"/><p>
    </c:if>
  </body>
</html>
```

The above code will generate the following result −

My salary is: 4000

Explain about JSP Standard Tag Library(JSTL).

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

Install JSTL Library

If we are using Apache Tomcat container then follow the following two simple steps:

- Download the binary distribution from Apache Standard Taglib and unpack the compressed file.
- To use the Standard Taglib from its Jakarta Taglibs distribution, simply copy the JAR files in the distribution's 'lib' directory to our application's webapps\ROOT\WEB-INF\lib directory.

To use any of the libraries, we must include a <taglib> directive at the top of each JSP that uses the library.

Core Tags

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

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

There are following Core JSTL Tags:

Tag	Description
<c:out >	Like <%= ... >, but for expressions.
<c:set >	Sets the result of an expression evaluation in a 'scope'
<c:remove >	Removes a scoped variable (from a particular scope, if specified).
<c:catch>	Catches any Throwable that occurs in its body and optionally exposes it.
<c:if>	Simple conditional tag which evalutes its body if the supplied condition is true.
<c:choose>	Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>

if action

The <c:if> tag evaluates an expression and displays its body content only if the expression evaluates to true.

Attribute

The <c:if> tag has the following attributes −

Attribute	Description	Required	Default
test	Condition to evaluate	Yes	None
var	Name of the variable to store the condition's result	No	None
scope	Scope of the variable to store the condition's result	No	page

Example

```
<%@ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c" %>
<html>
  <head>
    <title><c:if> Tag Example</title>
  </head>
  <body>
    <c:set var = "salary" scope = "session" value = "${2000*2}"/>
    <c:if test = "${salary > 2000}">
      <p>My salary is:  <c:out value = "${salary}"/><p>
    </c:if>
  </body>
</html>
```

The above code will generate the following result −

My salary is: 4000

Formatting tags

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

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

Following is the list of Formatting JSTL Tags:

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

The **<fmt:formatNumber>** tag has the following attributes

Attribute	Description	Required	Default
Value	Numeric value to display	Yes	None
type	NUMBER, CURRENCY, or PERCENT	No	Number
var	Name of the variable to store the formatted number	No	Print to page
scope	Scope of the variable to store the formatted number	No	page

- If the type attribute is percent or number, then you can use several number-formatting attributes. The **maxIntegerDigits** and **minIntegerDigits** attributes allow you to specify the size of the nonfractional portion of the number. If the actual number exceeds **maxIntegerDigits**, then the number is truncated.
- Attributes are also provided to allow you to determine how many decimal places should be used. The **minFractionalDigits** and **maxFractionalDigits** attributes allow you to specify the number of decimal places. If the number exceeds the maximum number of fractional digits, the number will be rounded.
- Grouping can be used to insert commas between thousands groups. Grouping is specified by setting the **groupingIsUsed** attribute to either true or false. When using grouping with **minIntegerDigits**, you must be careful to get your intended result.
- You may select to use the pattern attribute. This attribute lets you include special characters that specify how you would like your number encoded. Following table lists out the codes.

S.No.	Symbol & Description
1	**0** Represents a digit.
2	**E** Represents in exponential form.

3	# Represents a digit; displays 0 as absent.

Example

```
<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix = "fmt" uri = "http://java.sun.com/jsp/jstl/fmt" %>
<html>
  <head>
    <title>JSTL fmt:formatNumber Tag</title>
  </head>
  <body>
    <h3>Number Format:</h3>
    <c:set var = "balance" value = "120000.2309" />
    <p>Formatted Number (1): <fmt:formatNumber value = "${balance}"
      type = "currency"/></p>
    <p>Formatted Number (3): <fmt:formatNumber type = "number"
      maxFractionDigits = "3" value = "${balance}" /></p>

    <p>Formatted Number (5): <fmt:formatNumber type = "percent"
      maxIntegerDigits="3" value = "${balance}" /></p>
  </body>
</html>
```

The above code will generate the following result

Number Format:
Formatted Number (1): £120,000.23
Formatted Number (3): 120,000.231
Formatted Number (5): 023%

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 your JSP:

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

Following is the list of SQL JSTL Tags:

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

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 your 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" %>

Before you proceed with the examples, you would need to copy following two XML and XPath related libraries into your <Tomcat Installation Directory>\lib:

- **XercesImpl.jar:** Download it fromhttp://www.apache.org/dist/xerces/j/
- **xalan.jar:** Download it from http://xml.apache.org/xalan-j/index.html

Following is the list of XML JSTL Tags:

Tag	Description
<x:out>	Like <%= ... >, but for XPath expressions.
<x:parse>	Use to parse XML data specified either via an attribute or in the tag body.
<x:set >	Sets a variable to the value of an XPath expression.
<x:if >	Evaluates a test XPath expression and if it is true, it processes its body. If the test condition is false, the body is ignored.
<x:forEach>	To loop over nodes in an XML document.

The **<x:forEach>** tag has the following attributes −

Attribute	Description	Required	Default
select	The XPath expression to be evaluated	Yes	None
var	Name of the variable to store the current item for each loop	No	None
begin	The start index for the iteration	No	None
end	The end index for the iteration	No	None

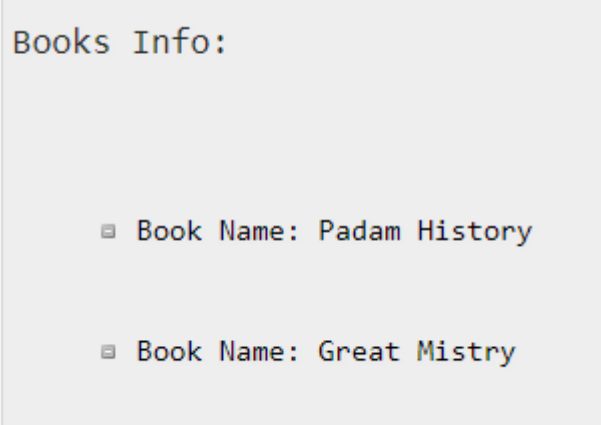
Example

The following example shows the use of the **<x:forEach>** tag −

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

<html>
  <head>
    <title>JSTL x:if Tags</title>
  </head>
  <body>
    <h3>Books Info:</h3>
    <c:set var = "xmltext">
      <books>
        <book>
          <name>Padam History</name>
          <author>ZARA</author>
          <price>100</price>
        </book>

        <book>
          <name>Great Mistry</name>
          <author>NUHA</author>
          <price>2000</price>
```

```
        </book>
      </books>
    </c:set>
    <x:parse xml = "${xmltext}" var = "output"/>
    <ul class = "list">
      <x:forEach select = "$output/books/book/name" var = "item">
        <li>Book Name: <x:out select = "$item" /></li>
      </x:forEach>
    </ul>
  </body>
</html>
```

You will receive the following result −



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 your JSP:

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

Following is the list of JSTL Functions:

Function	Description
fn:contains()	Tests if an input string contains the specified substring.
fn:containsIgnoreCase()	Tests if an input string contains the specified substring in a case insensitive way.
fn:endsWith()	Tests if an input string ends with the specified suffix.
fn:escapeXml()	Escapes characters that could be interpreted as XML markup.
fn:indexOf()	Returns the index withing a string of the first occurrence of a specified substring.
fn:join()	Joins all elements of an array into a string.
fn:replace()	Returns a string resulting from replacing in an input string all occurrences with a given string.
fn:split()	Splits a string into an array of substrings.

fn:startsWith()	Tests if an input string starts with the specified prefix.

The **fn:split()** function splits a string into an array of substrings based on a delimiter string.

Syntax

The **fn:split()** function has the following syntax

java.lang.String[] split(java.lang.String, java.lang.String)

Example

Following is the example to explain the functionality of the **fn:split()** function

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

<html>
  <head>
    <title>Using JSTL Functions</title>
  </head>

  <body>
    <c:set var = "string1" value = "This is first String."/>
    <c:set var = "string2" value = "${fn:split(string1, ' ')}" />
    <c:set var = "string3" value = "${fn:join(string2, '-')}" />

    <p>String (3) : ${string3}</p>

    <c:set var = "string4" value = "${fn:split(string3, '-')}" />
    <c:set var = "string5" value = "${fn:join(string4, ' ')}" />

    <p>String (5) : ${string5}</p>
  </body>
</html>
```

You will receive the following result

String (3) : This-is-first-String.
String (5) : This is first String.

Jsp and database connectivity:

We are going to create jsp registration form database. We first create simple registration form. All data will be stored in he database.

We use method getParameter() that return the value of a request parameter passed as string of request. We have used database connection in this form. We can insert, update, delete any data in this form. This application can be used by any jsp web project.

syntax of getParameter:-

```
<%= request.getParameter("massage")%>
```

Registration.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<%@page import="java.sql.Statement"%>
<%@page import="java.sql.Connection"%>
<%@page import="java.sql.DriverManager"%>
<%@page import="java.sql.ResultSet"%>
```

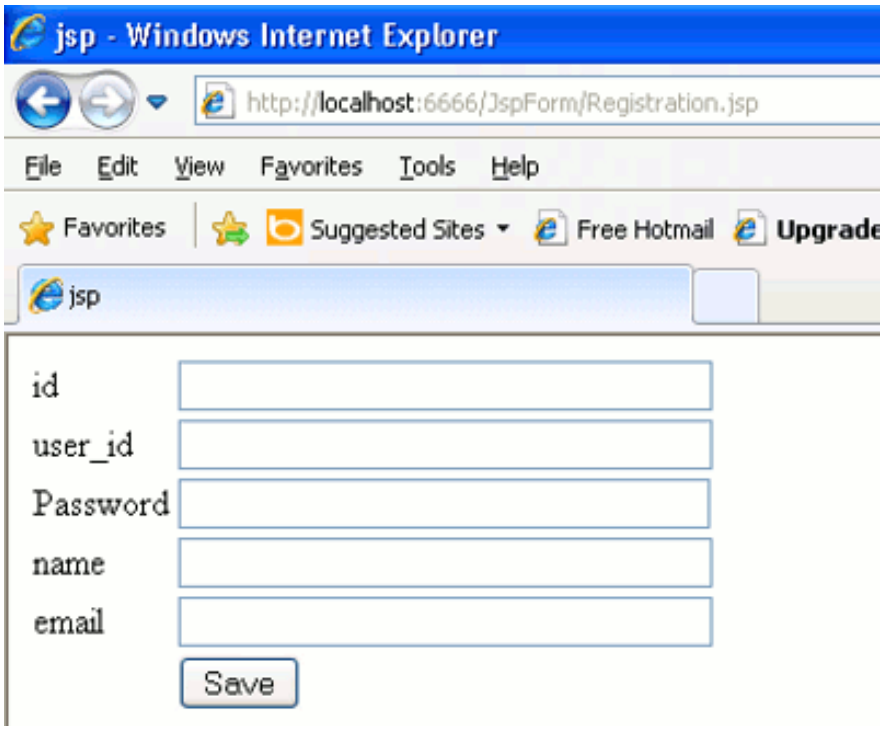
```
<%
String id = "";
String user_id = request.getParameter("user_name");
String pwd = request.getParameter("pwd");
String name = request.getParameter("name");
String email = request.getParameter("email");
if (!(user_id == null || user_id.isEmpty())) {

String driverName = "com.mysql.jdbc.Driver";
String connectionUrl = "jdbc:mysql://localhost:3306/";
String dbName = "naulej";
String userId = "root";
String password = "root";
try {
Class.forName(driverName);
}
 catch (ClassNotFoundException e) {
e.printStackTrace();
}

Connection connection = null;
Statement statement = null;
try {
connection = DriverManager.getConnection(connectionUrl + dbName, userId, password);
statement = connection.createStatement();
String sql = "INSERT INTO userinfo (user_id,pwd,name,email) VALUES('"+ user_id + "','" + pwd + "','" +
                              name + "','" + email + "')";
int flage = statement.executeUpdate(sql);
}
 catch (Exception e) {
e.printStackTrace();
}
}
%>
```

Index.html

```
<html>
<head>
<head>
<title>jsp</title>
</head>
<form method="post" action="Registration.jsp">
<table>
<tr>
<td>id</td><td><input type="text" id="id" size="35" /></td>
</tr>
<tr>
<td>user_id</td>
<td><input type=text name="user_name" size="35" /></td>
</tr>
<tr>
```

```
<td>Password</td>
<td><input type="password" name="pwd" size="35" /></td>
</tr>
<tr>
<td>name</td>
<td><input type="text" name="name" size="35" /></td>
</tr>
<tr>
<td>email</td>
<td><input type="text" name="email" size="35" /></td>
</tr>
<tr>
<td></td>
<td><input type="submit" name="insert" value="Save"></td>
</tr>
</table>
</form>
</html>
```

Output:



JavaBean in JSP:

A JavaBean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.

Following are the unique characteristics that distinguish a JavaBean from other Java classes −

- It provides a default, no-argument constructor.
- It should be serializable and that which can implement the **Serializable** interface.
- It may have a number of properties which can be read or written.
- It may have a number of "**getter**" and "**setter**" methods for the properties.

JavaBeans Properties

A JavaBean property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define.

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

S.No.	Method & Description
1	get**PropertyName**() For example, if property name is *firstName*, your method name would be **getFirstName()** to read that property. This method is called accessor.
2	set**PropertyName**() For example, if property name is *firstName*, your method name would be **setFirstName()** to write that property. This method is called mutator.

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

JavaBeans Example

Consider a student class with few properties −

```
package com.tutorialspoint;

public class StudentsBean implements java.io.Serializable {
  private String firstName = null;
  private String lastName = null;
  private int age = 0;

  public StudentsBean() {
  }
  public String getFirstName(){
    return firstName;
  }
  public String getLastName(){
    return lastName;
  }
  public int getAge(){
    return age;
  }
  public void setFirstName(String firstName){
    this.firstName = firstName;
  }
  public void setLastName(String lastName){
    this.lastName = lastName;
  }
  public void setAge(Integer age){
    this.age = age;
  }
}
```

Accessing JavaBeans

The **useBean** action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows −

```
<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>
```

Here values for the scope attribute can be a **page, request, session** or **application based** on your requirement. The value of the **id** attribute may be any value as a long as it is a unique name among other **useBean declarations** in the same JSP.

Following example shows how to use the useBean action

```
<html>
  <head>
    <title>useBean Example</title>
  </head>

  <body>
    <jsp:useBean id = "date" class = "java.util.Date" />
    <p>The date/time is <%= date %>
  </body>
</html>
```

You will receive the following result

The date/time is Thu Sep 30 11:18:11 GST 2010

Accessing JavaBeans Properties

Along with **<jsp:useBean...>** action, you can use the **<jsp:getProperty/>** action to access the get methods and the **<jsp:setProperty/>** action to access the set methods. Here is the full syntax −

```
<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">
  <jsp:setProperty name = "bean's id" property = "property name"
    value = "value"/>
  <jsp:getProperty name = "bean's id" property = "property name"/>
  .....
</jsp:useBean>
```

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the **get** or the **set** methods that should be invoked.

Following example shows how to access the data using the above syntax −

```
<html>
  <head>
    <title>get and set properties Example</title>
  </head>

  <body>
    <jsp:useBean id = "students" class = "com.tutorialspoint.StudentsBean">
      <jsp:setProperty name = "students" property = "firstName" value = "Zara"/>
      <jsp:setProperty name = "students" property = "lastName" value = "Ali"/>
      <jsp:setProperty name = "students" property = "age" value = "10"/>
    </jsp:useBean>

    <p>Student First Name:
      <jsp:getProperty name = "students" property = "firstName"/>
    </p>

    <p>Student Last Name:
      <jsp:getProperty name = "students" property = "lastName"/>
    </p>

    <p>Student Age:
      <jsp:getProperty name = "students" property = "age"/>
    </p>
```

```
  </body>
</html>
```

Result will be

Student First Name: Zara
Student Last Name: Ali
Student Age: 10

JSP object

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
Exception	The Exception object allows the exception data to be accessed by designated JSP.

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.

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.

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. Following are the important methods which we would use to write boolean char, int, double, object, String etc.

TWO MARK

1. What is JavaScript?

JavaScript is a platform-independent, event-driven, interpreted client-side scripting language developed by Netscape Communications Corp. and Sun Microsystems.

2. What are the primitive data types in javascript?

JavaScript supports five primitive data types: number, string, Boolean, undefined, and null. These types are referred to as primitive types because they are the basic building blocks from which more complex types can be built. Of the five, only number, string, and Boolean are real data types in the sense of actually storing data.

Undefined and null are types that arise under special circumstances.

3. What are the Escape Codes Supported in JavaScript?

The Escape codes supported in javascript are \b Backspace,\t Tab (horizontal), \n Linefeed (newline),\v Tab (vertical),\f Form feed,\r Carriage return,\" Double quote \' Single quote,\\ Backslash.

4. What is JavaScript name spacing? How and where is it used?

Using global variables in JavaScript is evil and a bad practice. That being said, namespacing is used to bundle up all your functionality using a unique name. In JavaScript, a namespace is really just an object that you've attached all further methods, properties and objects. It promotes modularity and code reuse in the application.

5. How many looping structures can you find in javascript?

If you are a programmer, you know the use of loops. It is used to run a piece of code multiple times according to some particular condition. Javascript being a popular scripting language supports the following loops for, while, do-while loop

6. Mention the various Java Script Object Models.

Math Object, String Object, Date Object, Boolean and Number Object, Document Object Window Object.

7. How Scripting Language Is Differs from HTML?

HTML is used for simple web page design, HTML with FORM is used for both form design and Reading input values from user, Scripting Language is used for Validating the given input values weather it is correct or not, if the input value is incorrect, the user can pass an error message to the user, Using form concept various controls like Text box, Radio Button, Command Button, Text Area control and List box can be created.

8. What are JSP actions?

JSP actions use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.

9. Justify "JavaScript" is an event-driven programming"

Javascript supports event driven programming. when user clicks the mouse or hit the keys on the keyboard or if user submits the form then these events and response to them can be handled using javascript. Hence javascript is mainly used in web programming for validating the data provided by the user.

10. What is the use of pop up boxes in java script?

There are three types of popup boxes used in javascript. Using these popup boxes the user can interact with the web application.

11. What is DOM?

Document Object Model (DOM) is a set of platform independent and language neutral application interface (API) which describes how to access and manipulate the information stored in XML, XHTML and javascript documents.

12. Enlist any four mouse events.

The MouseEvent are-mousedown, mouseup, mouseover, mousemove, mouseout.

13. List ad various level of document object modeling.

Various levels of DOM are DOM0, Dom1, Dom2, and Dom3

14. What are they validation properties and methods?

Validation properties and methods are checkvalidity (), validaionMessage, customerror, patternMismatch, rangeOverflow, rangeUnderflow, tooLong.

15. Define event bubbling.

Suppose, there is an element present inside another element. Then during the event handling, if the event which is present in the inner element is handled and then the event of the outer element is handled. This process of event handling is called event bubbling

16. How to create arrays in Javascript?

We can declare an array like this Var scripts = new Array();
We can add elements to this array like this
scripts[0] = "PHP";
scripts[1] = "ASP";
scripts[2] = "JavaScript";
scripts[3] = "HTML";

Now our array scrips has 4 elements inside it and we can print or access them by using their index number. Note that index number starts from 0. To get the third element of the array we have to use the index number 2. Here is the way to get the third element of an array. document. write (scripts[2]); We also can create an array like this var no_array = new Array(21, 22, 23, 24, 25);

17. Write a simple program in JavaScript to validate the email-id.

```
<!DOCTYPE html>
<html>
<head>
<script>
function validateForm() {
var x = document.forms["myForm"]["email"].value;
var atpos = x.indexOf("@");
var dotpos = x.lastIndexOf(".");
if (atpos<1 || dotpos<atpos+2 || dotpos+2>=x.length) {
alert("Not a valid e-mail address");
return false;}}
</script>
</head>
<body>
<form name="myForm" action="demo_form.asp" onsubmit="return validateForm();" method="post">
Email: <input type="text" name="email">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

18. Write short notes on JDBC.

JDBC standard is intented for people developing industrial-strength database applications.JDBC makes java effective for developing enterprise information system.java.sql is the JDBC package that contains classes & interfaces that enable a java program to interact with a database.

19. Write short notes on JDBC drivers.

A JDBC driver is basically an implementation of the function calls specified in the JDBC API for a particular vendor's RDBMS. Hence, a java program with JDBC function calls can access any RDBMS that has a JDBC driver available. A driver manager is used to keep track of all the installed drivers on the system. The operations of driver manager are getDriver, registerDriver, deregisterDriver.

20. What are the advantages of servlet over CGI?

Servlets are easier to write.
Servlets are faster to run.
Servlets are platform independent. CGI has the disadvantage of doing server-side programming with platform-specific APIs.
Servlet can handle multiple request concurrently

21. Write down the methods of servlet interface

void destroy() –called when the servlet is unloaded.
ServletConfig getServletConfig() –returns a ServletConfig object that contains any initialization parameters.
String get ServletInfo() – returns a string describing the servlet.
void init(ServletConfig sc) throws ServletException –called when the servlet is

initialized .Initialization parameters for servlet can be obtained from sc. An unavailable exception should be thrown if the servlet is not initialized.
Void Service(ServletRequest req,ServletResponse res) throws ServletException, IOException- Called to process a request from a client. The request from the client can be read from req. response to the client can be written to res.
An exception is generated if a servlet or IO problem occurs.

22. What is a Servlet?

Java Servlets are server side components that provides a powerful mechanism for developing server side of web application. Earlier CGI was developed to provide server side capabilities to the web applications. Although CGI played a major role in the explosion of the Internet, its performance, scalability and reusability issues make it less than optimal solutions. Java Servlets changes all that. Built from ground up using Sun's write once run anywhere technology java servlets provide excellent framework for server side processing.

21. Write down the methods of servlet interface

void destroy() –called when the servlet is unloaded.
ServletConfig getServletConfig() –returns a ServletConfig object that contains any initialization parameters.
String get ServletInfo() – returns a string describing the servlet.
void init(ServletConfig sc) throws ServletException –called when the servlet is
initialized .Initialization parameters for servlet can be obtained from sc. An unavailable exception should be thrown if the servlet is not initialized.
Void Service(ServletRequest req,ServletResponse res) throws ServletException, IOException- Called to process a request from a client. The request from the client can be read from req. response to the client can be written to res.
An exception is generated if a servlet or IO problem occurs.

22. What is the difference between CGI and servlets?

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 %>.

25. What are advantages of using JSP?

- It does not require advanced knowledge of JAVA
- It is capable of handling exceptions
- Easy to use and learn
- It can tags which are easy to use and understand
- Implicit objects are there which reduces the length of code
- It is suitable for both JAVA and non JAVA programmer

26. Explain lifecycle of a JSP.

- Compilation
- Initialization
- Execution
- Cleanup

27. What are the types of directive tags?

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

28. Differentiate between doGet and doPost method?

doGet is used when there is are requirement of sending data appended to a query string in the URL. The doGet models the GET method of Http and it is used to retrieve the info on the client from some server as a request to it. The doGet cannot be used to send too much info appended as a query stream. GET puts the form values into the URL string. GET is limited to about 256 characters (usually a browser limitation) and creates really ugly URLs.
doPost allows you to have extremely dense forms and pass that to the server without clutter or limitation in size. e.g. you obviously can't send a file from the client to the server via GET. POST has no limit on the amount of data you can send and because the data does not show up on the URL you can send passwords. But this does not mean that POST is truly secure. For real security you have to look into encryption which is an entirely different topic

29. Why there are no constructors in servlets?

A servlet is just like an applet in the respect that it has an init() method that acts as a constrcutor. Since the servlet environment takes care of instantiating the servlet, an explicit constructor is not needed. Any initialization code you need to run should be placed in the init() method since it gets called when the servlet is first loaded by the servlet container.

30. What is the difference between GenericServlet and HttpServlet?

GenericServlet is for servlets that might not use HTTP, like for instance FTP servlets. Of course, it turns out that there's no such thing as FTP servlets, but they were trying to plan for future growth when they designed the spec. Maybe some day there will be another subclass, but for now, always use HttpServlet.

31. How much data we can store in a session object?

- Any amount of data can be stored there because the session is kept on the server side.
- The only limitation is sessionId length, which shouldn't exceed ~4000 bytes – this limitation is implied by HTTP header length limitation to 4Kb since sessionId may be stored in the cookie or encoded in URL (using "URL rewriting") and the cookie specification says the size of cookie as well as HTTP request (e.g. GET /document.html\n) cannot be longer then 4kb.

32. What is the need for tracking session?

Http is a stateless protocol, so there is no way for a server to recognize that a sequence of requests is all from same client.

Eg:

On-line shopping.

33. List the disadvantages of using cookies.

- Some browser doesn‘t support cookies.
- Due to limited memory space, Wireless Application Protocol (WAP) gateways don‘t accept cookies.
- Any intruder (unauthorized person) using client‘s machine can change value of cookies.

34. What is Session ID? A session ID is a unique identification

string usually a long, random and alpha-numeric string, that is transmitted between the client and the server. Session IDs are usually stored in the cookies, URLs (in case url rewriting) and hidden fields of Web pages.

UNIT - III

JDBC Program Example

For executing following programs -

1) JDK must be installed
2) XAMPP must be installed
3) Eclipse must be installed. Also add jar file mysql-connector-java. Get it downloaded from Internet and save it at suitable location. Then configure Eclipse to use mySQL, for that purpose.
   i) Right click on your project.
   ii) Click on properties -> Java Build path -> Libraries
   iii) Click Add External JARs... button and select mysql-connector-java xxx.jar file

example Consider the a database table with the following schema. (PNR No, status). Write a servlet program to display the status, given the PNR number.

Sol.: Java Program [PNRDemo.java]

```
import java.io.*;
import java.util.*;
import javax.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

```
Public class PNRDemo extends HttpServlet.
{
  public void service(HttpServlet Request request, HttpServlet
  Response response)
  throws IOException, ServletException
  {
  // Connecting to database
  Connection con = null;
  Statement stmt = null;
  ResultSet rs = null;
  Print Writer out = response.getWriter();
try
{
class.forName("Sun.jdbc.odbc.odbc.jdbcodbc
                                    Driver");
Con = DriverManager.getConnection("jdbc:odbc:
                                    PNRDemo","","");
}
 Catch (ClassNotFoundException e)
{ e.printStackTrace();}
Catch (SQLException e)
{ e.printStackTrace();}
Catch (Exception e)
{ e.printStackTrace();}
try
{
   stmt = con.createStatement();
  rs = stmt.executeQuery("SELECT* FROM PNR_
                                    table");
 // displaying records
```

```
response.setContentType("text/html");
Out.println("<html>");
Out.println("<head><title>PNR status Display</title>
                                        </head>");
Out.println("<body>");
Out.print("<center>");
Out.print("<h2> PNR Details</h2>");
Out.print("<table border =3>");
Out.print("<th> PNR No. </th>");
Out.print("<th> Status </th>");
while(rs.next())
{
Out.print("<tr>");
Out.print("<td>");
Out.print(rs.getObject(1).toString());
Out.print("</td>");
Out.print("</tr>");
}
Out.print("</table>");
Out.print("</center>");
Out.println("</body></html>");
}
Catch(SQLException e)
{
}
finally {
  try {
```

```
if (rs != null)
{
rs.close();
rs = null;
}
if (stmt != null)
{
stmt.close();
stmt = null;
}
if (con != null) {
con.close();
con = null;
}
} catch (SQLException e) {}
}
out.close();
} // end of service function
} // end of class
```

Output

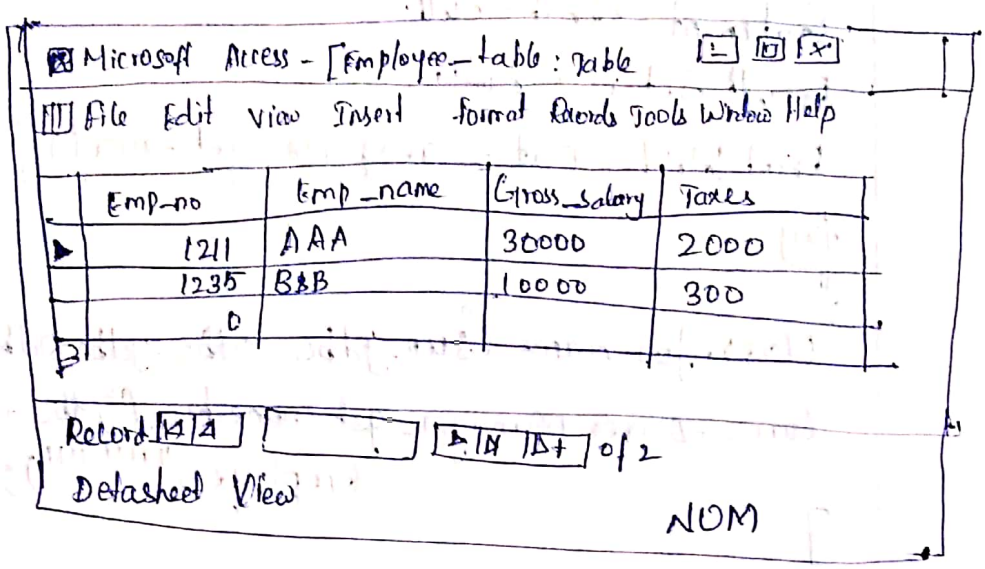
PNR Details

PNR No.	Status
10	aaa
20	bbb
30	ccc

Example

Write a Java Servlet to display Net salary of employee, Use JDBC Connectivity to get employee details from database

Sol: step 1 : Create a employees database as follows -



Step 2: The servlet for Computing the net salary is as given below

```
import java.io.*;
import java.util.*;
import javax.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class empdbase extends HttpServlet
{
  public void service(HttpServletRequest request,
```

```
{
// Connecting to database
Connection con=null;
Statement stmt=null;
Result Set rs =null;
Print Writer Out = response.getWriter();
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
con=DriverManager.getconnection("jdbc:odbc:
                                Employees","","");
}
Catch (ClassNotFound Exception e)
{ e.printStackTrace(); }
Catch (SQLException e)
{e.printStackTrace();}
Catch(Exception e)
{e.printStackTrace();}
try
{
stmt = con.createStatement();
rs = stmt.executeQuery ("SELECT * FROM
                         Employee_table");
// displaying records
```

```
response.setContentType("text/html");
    out.println("<html>");
out.println("<head><title>Servlet Database
            Connectivity</title></head>");
out.println("<body>");

out.print("<center>");
out.print("<h2>Employees Database</h2>");
out.print("<table border=3>");
out.print("<th>Emp_ID</th>");
out.print("<th>Name</th>");
out.print("<th>Gross</th>");
out.print("<th>Taxes</th>");
out.print("<th>Net Salary</th>");
while(rs.next())
{
  int tot=0;
  string gross=rs.getstring(3);
  string deductions=rs.getstring(4);

  int g=Integer.parseInt(gross);
  int d=Integer.parseInt(deductions);

tot=g-d; //net salary of employee
    out.print("<tr>");
   out.print("<td>");
   out.print(rs.getInt(1));
```

```
        out.print("</td>");
        out.print("<td>");
        out.print(rs.getString(2));
      out.print("</td>");
      out.print("<td>");
        out.print(g);
        out.print("</td>");
        out.print("<td>");
        out.print(d);
      out.print("</td>");
      out.print("<td>");
      out.println(tot);
      out.print("</td>");
       out.print("</tr>");
  }
out.print("</table>");
out.print("</center>");
out.println("</body></html>");
}
catch(SQLException e){}
finally{
try{
if(rs!=null){
}
rs.close();
  rs=null;
}
```

```
if (stmt != null)
{
stmt.close();
 stmt=null;
}
if (con != null)
{
 con.close();
 con=null;
}
}
catch (SQLException e) {}
}
out.close();
} //end of service function
} // end of class
```