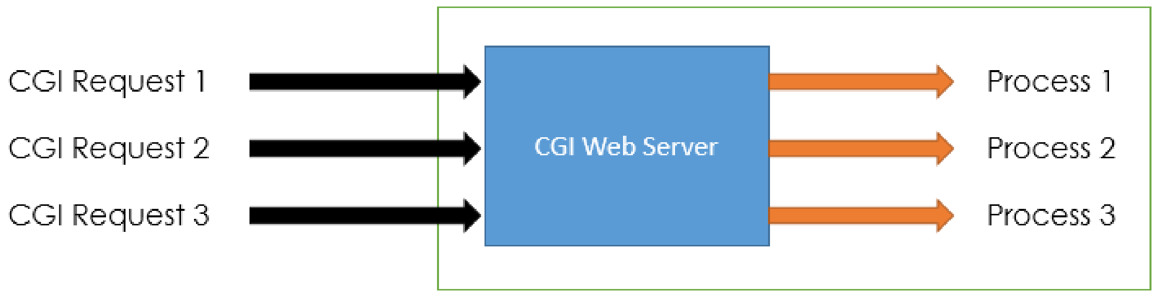
UNIT IV

CGI (Commmon Gateway Interface)

1. CGI (Common Gateway Interface) is used to provide dynamic content to the user.
2. CGI is used to execute a program that resides in the server to process data or access databases to produce the relevant dynamic content.
3. Programs that resides in server can be written in native operating system such as C++.

Diagrammatic Representation :



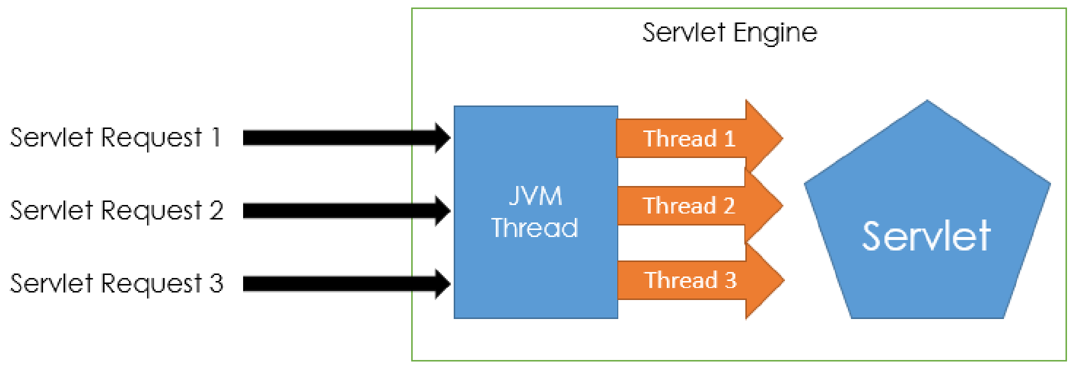
We have listed some problems in CGI technology –

Disadvantages of CGI :

1. For each request CGI Server receives, It creates new Operating System Process.
2. If the number of requests from the client increases then more time it will take to respond to the request.
3. As programs executed by CGI Script are written in the native languages such as C, C++, perl which are platform independent.

Servlet :

CGI programs are used to execute programs written inside the native language. But in Servlet all the programs are compiled into the Java bytecode which is then run in the Java virtual machine.



In Servlet, All the requests coming from the Client are processed with the threads instead of the OS process.

Servlet Vs CGI :

Let's differentiate Servlet and CGI –

Servlet	**CGI (Common Gateway Interface)**
Servlets are portable	CGI is not portable.
In Servlets each request is handled by lightweight Java Thread	IN CGI each request is handled by heavy weight OS process
In Servlets, Data sharing is possible	In CGI, data sharing is not available.
Servlets can link directly to the Web server	CGI cannot directly link to Web server.
Session tracking and caching of previous computations can be performed	Session tracking and caching of previous computations cannot be performed
Automatic parsing and decoding of HTML form data can be performed.	Automatic parsing and decoding of HTML form data cannot be performed.
Servlets can read and Set HTTP Headers	CGI cannot read and Set HTTP Headers
Servlets can handle cookies	CGI cannot handle cookies

Servlet	**CGI (Common Gateway Interface)**
Servlets can track sessions	CGI cannot track sessions
Servlets is inexpensive than CGI	CGI is more expensive than Servlets

Static vs Dynamic website

Static Website	**Dynamic Website**
Prebuilt content is same every time the page is loaded.	Content is generated quickly and changes regularly.
It uses the **HTML** code for developing a website.	It uses the server side languages such as **PHP,SERVLET, JSP, and ASP.NET** etc. for developing a website.
It sends exactly the same response for every request.	It may generate different HTML for each of the request.
The content is only changed when someone publishes and updates the file (sends it to the web server).	The page contains "server-side" code which allows the server to generate the unique content when the page is loaded.
Flexibility is the main advantage of static website.	Content Management System (CMS) is the main advantage of dynamic website.

Java servlet:

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

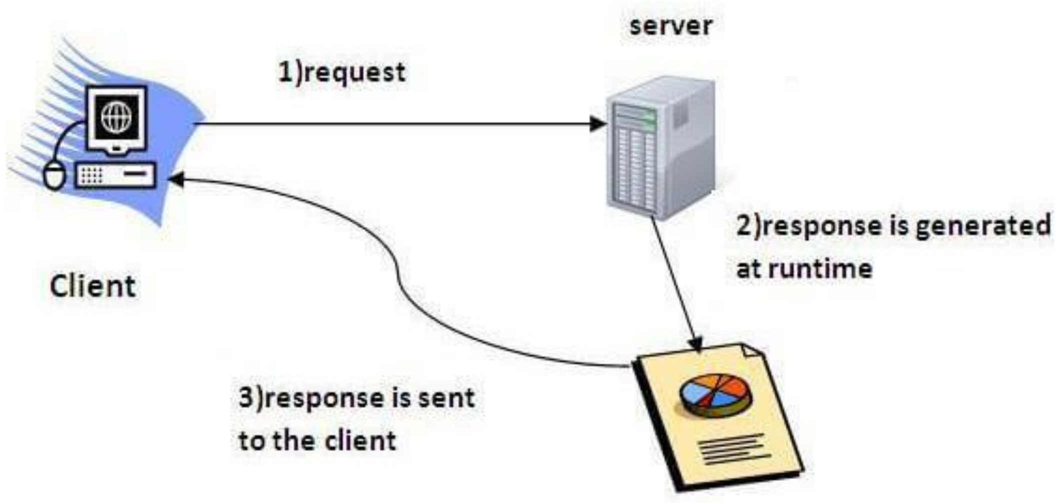
Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language.

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

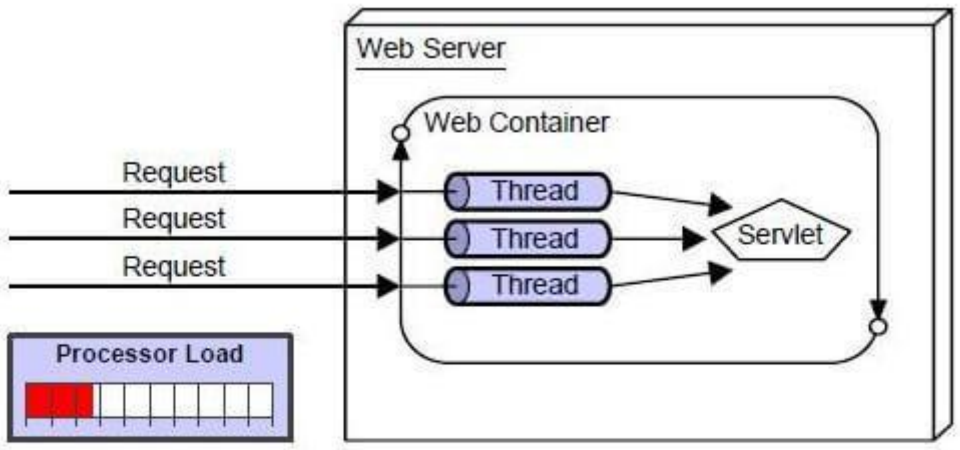
What is a Servlet?

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

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


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

Benifits of Servlet



There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages of Servlet are as follows:

1. **Better performance:** because it creates a thread for each request, not process.
2. **Portability:** because it uses Java language.

3. **Robust:** JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.
4. **Secure:** because it uses java language.

Life cycle of java servlet:

Life Cycle of a Servlet

The entire life cycle of a Servlet is managed by the Servlet container which uses the **javax.servlet.Servlet** interface to understand the Servlet object and manage it. So, before creating a Servlet object let's first understand the life cycle of the Servlet object which is actually understanding that how the Servlet container manages the Servlet object.

Stages of the Servlet Life Cycle: The Servlet life cycle mainly goes through four stages,

Loading a Servlet.

Initializing the Servlet.

Request handling.

Destroying the Servlet.

Let's look at each of these stages in details:

Loading a Servlet: The first stage of the Servlet lifecycle involves loading and initializing the Servlet by the Servlet container. The Web container or Servlet Container can load the Servlet at either of the following two stages :

-Initializing the context, on configuring the Servlet with a zero or positive integer value.

-If the Servlet is not preceding stage, it may delay the loading process until the Web container determines that this Servlet is needed to service a request.

The Servlet container performs two operations in this stage :

Loading : Loads the Servlet class.

Instantiation : Creates an instance of the Servlet. To create a new instance of the Servlet, the container uses the no-argument constructor.

Initializing a Servlet: After the Servlet is instantiated successfully, the Servlet container initializes the instantiated Servlet object. The container initializes the Servlet object by invoking the **Servlet.init(ServletConfig)** method which accepts ServletConfig object reference as parameter.

The Servlet container invokes the **Servlet.init(ServletConfig)** method only once, immediately after the **Servlet.init(ServletConfig)** object is instantiated successfully. This method is used to initialize the resources, such as JDBC datasource.

Now, if the Servlet fails to initialize, then it informs the Servlet container by throwing the **ServletException** or **UnavailableException.**

Handling request: After initialization, the Servlet instance is ready to serve the client requests. The Servlet container performs the following operations when the Servlet instance is located to service a request :

It creates the **ServletRequest** and **ServletResponse** objects. In this case, if this is a HTTP request then the Web container creates **HttpServletRequest** and **HttpServletResponse** objects which are subtypes of the **ServletRequest** and **ServletResponse** objects respectively.

After creating the request and response objects it invoke the Servlet.service(ServletRequest, ServletResponse) method by passing the request and response objects.

The **service()** method while processing the request may throw the **ServletException** or **UnavailableException** or **IOException.**

Destroying a Servlet: When a Servlet container decides to destroy the Servlet, it performs the following operations,

It allows all the threads currently running in the service method of the Servlet instance to complete their jobs and get released.

After currently running threads have completed their jobs, the Servlet container calls the **destroy()** method on the Servlet instance.

After the **destroy()** method is executed, the Servlet conatiner releases all the references of this Servlet instance so that it becomes eligible for garbage collection.

Working with cookies:

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 a 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 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 provide
other interfaces. They are:

1. **public void addCookie(Cookie ck):**method of HttpServletResponse interface is u
   to add cookie in response object.
2. **public Cookie[] getCookies():**method of HttpServletRequest interface is use
   return all the cookies from the browser.

How to create 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
```

How to delete 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
```

How to get Cookies?

Let's see the simple code to get all the cookies.

```
Cookie ck[]=request.getCookies();
for(int i=0;i<ck.length;i++){
 out.print("<br>"+ck[i].getName()+" "+ck[i].getValue());//printing name and value of cookie
}
```

Simple example of Servlet Cookies

In this example, we are storing the name of the user in the cookie object and accessing it in another servlet. As we know well that session corresponds to the particular user. So if you access it from too many browsers with different values, you will get the different value.

index.html

```
<form action="servlet1" method="post">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
  public class FirstServlet extends HttpServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse response){
    try{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String n=request.getParameter("userName");
    out.print("Welcome "+n);
    Cookie ck=new Cookie("uname",n);//creating cookie object
    response.addCookie(ck);//adding cookie in the response
    //creating submit button
```

```
 out.print("<form action='servlet2'>");
 out.print("<input type='submit' value='go'>");
 out.print("</form>");
 out.close();
   }catch(Exception e){System.out.println(e);}
} }
```

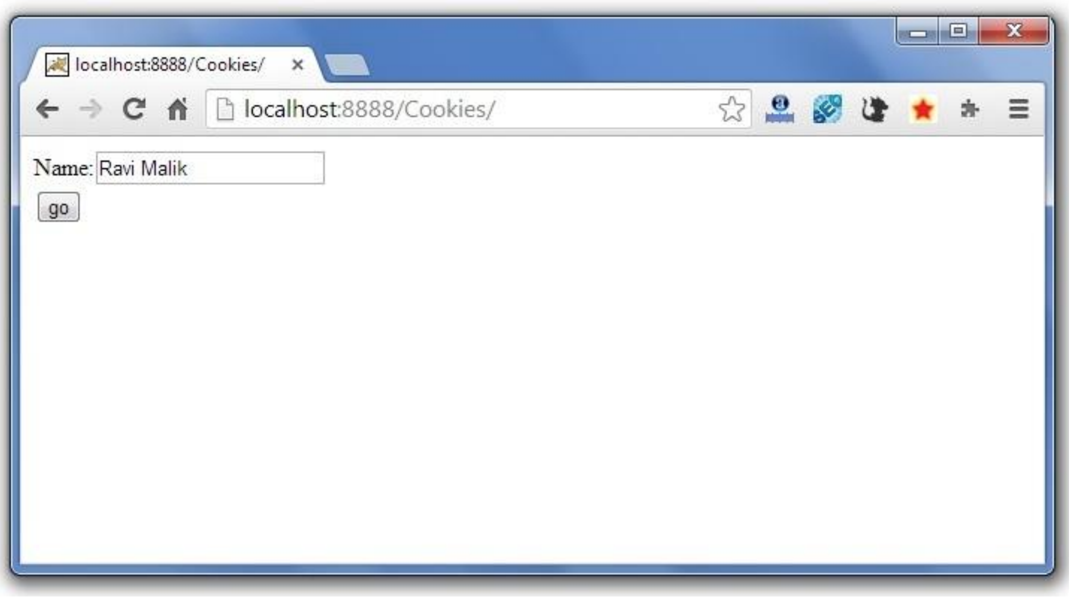
SecondServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
  public class SecondServlet extends HttpServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse response){
    try{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    Cookie ck[]=request.getCookies();
    out.print("Hello "+ck[0].getValue());
  out.close();
      }catch(Exception e){System.out.println(e);}
    }
  }
```

web.xml

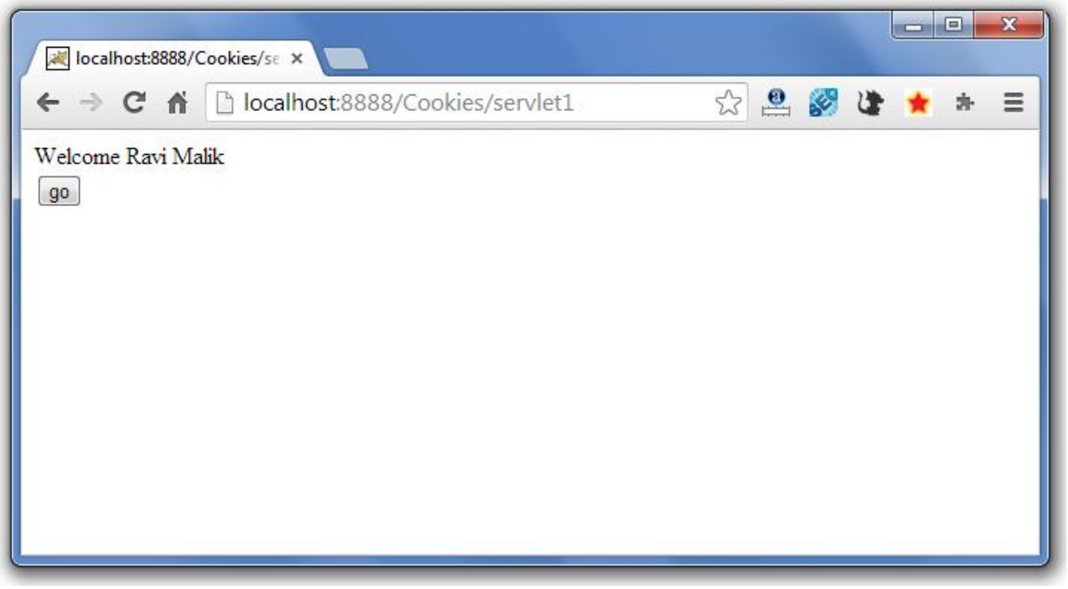
```
<web-app>
  <servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>
  <servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>
  <servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>
  <servlet-mapping>
```

```
15. <servlet-name>s2</servlet-name>
16. <url-pattern>/servlet2</url-pattern>
17. </servlet-mapping>
18.   </web-app>
```


localhost:8888/Cookies/

Name: Ravi Malik

go


localhost:8888/Cookies/servlet1

Welcome Ravi Malik

go

Session Tracking in Servlets

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 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.

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

Why use Session Tracking?

To recognize the user It is used to recognize the particular user.

Session Tracking Techniques

There are four techniques used in Session tracking:

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

2) Hidden Form Field

In case of Hidden Form Field **a hidden (invisible) textfield** is used for maintaining the state of an user.

In such case, we store the information in the hidden field and get it from another servlet. This approach is better if we have to submit form in all the pages and we don't want to depend on the browser.

Let's see the code to store value in hidden field.

```
1. <input type="hidden" name="uname" value="Vimal Jaiswal">
```

Here, uname is the hidden field name and Vimal Jaiswal is the hidden field value.

Real application of hidden form field

It is widely used in comment form of a website. In such case, we store page id or page name in the hidden field so that each page can be uniquely identified.

Advantage of Hidden Form Field

1. It will always work whether cookie is disabled or not.

Disadvantage of Hidden Form Field:

1. It is maintained at server side.
2. Extra form submission is required on each pages.
3. Only textual information can be used.

Example of using Hidden Form Field

In this example, we are storing the name of the user in a hidden textfield and getting that value from another servlet.

index.html

```
1. <form action="servlet1">
```

```
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
 public class FirstServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response){
        try{
         response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String n=request.getParameter("userName");
        out.print("Welcome "+n);
        //creating form that have invisible textfield
        out.print("<form action='servlet2'>");
        out.print("<input type='hidden' name='uname' value='"+n+"'>");
        out.print("<input type='submit' value='go'>");
        out.print("</form>");
        out.close();
              }catch(Exception e){System.out.println(e);}
   }
 }
```

SecondServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SecondServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
        try{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        //Getting the value from the hidden field
        String n=request.getParameter("uname");
        out.print("Hello "+n);
        out.close();
                }catch(Exception e){System.out.println(e);}
```

```
    }
}
```

web.xml

```
<web-app>
 <servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>
 <servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>
 <servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>
 <servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>
  </web-app>
```

3)URL Rewriting

In URL rewriting, we append a token or identifier to the URL of the next Servlet or the next resource. We can send parameter name/value pairs using the following format:

url?name1=value1&name2=value2&??

A name and a value is separated using an equal = sign, a parameter name/value pair is separated from another parameter using the ampersand(&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server. From a Servlet, we can use getParameter() method to obtain a parameter value.

Advantage of URL Rewriting

1. It will always work whether cookie is disabled or not (browser independent).
2. Extra form submission is not required on each pages.

Disadvantage of URL Rewriting

1. It will work only with links.
2. It can send Only textual information.

Example of using URL Rewriting

In this example, we are maintaning the state of the user using link. For this purpose, we are appending the name of the user in the query string and getting the value from the query string in another page.

index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class FirstServlet extends HttpServlet {
 public void doGet(HttpServletRequest request, HttpServletResponse response){
        try{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String n=request.getParameter("userName");
        out.print("Welcome "+n);
```

```
      //appending the username in the query string
       out.print("<a href='servlet2?uname="+n+"'>visit</a>");
        out.close();
            }catch(Exception e){System.out.println(e);}
  }  }
```

SecondServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
 public class SecondServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
       try{
       response.setContentType("text/html");
       PrintWriter out = response.getWriter();
       //getting value from the query string
       String n=request.getParameter("uname");
       out.print("Hello "+n);
        out.close();
              }catch(Exception e){System.out.println(e);}
 }  }
```

web.xml

```
<web-app>
  <servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>
  <servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>
  <servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>
 <servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>
```

```
18.   </web-app>
```

4) HttpSession interface

In such case, container creates a session id for each user.The container uses this id to identify the particular user.An object of HttpSession can be used to perform two tasks:

1. bind objects
2. view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.

How to get the HttpSession object ?

The HttpServletRequest interface provides two methods to get the object of HttpSession:

1. **public HttpSession getSession():**Returns the current session associated with this request, or if the request does not have a session, creates one.
2. **public HttpSession getSession(boolean create):**Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

Commonly used methods of HttpSession interface

1. **public String getId():**Returns a string containing the unique identifier value.
2. **public long getCreationTime():**Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
3. **public long getLastAccessedTime():**Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
4. **public void invalidate():**Invalidates this session then unbinds any objects bound to it.

Example of using HttpSession

In this example, we are setting the attribute in the session scope in one servlet and getting that value from the session scope in another servlet. To set the attribute in the session scope, we have used the setAttribute() method of HttpSession interface and to get the attribute, we have used the getAttribute method.

index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
 public class FirstServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response){
       try{
        response.setContentType("text/html");
       PrintWriter out = response.getWriter();
        String n=request.getParameter("userName");
       out.print("Welcome "+n);
        HttpSession session=request.getSession();
       session.setAttribute("uname",n);
     out.print("<a href='servlet2'>visit</a>");
            out.close();
          }catch(Exception e){System.out.println(e);}
   }  }
```

SecondServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
 public class SecondServlet extends HttpServlet {
```

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
        try{
          response.setContentType("text/html");
        PrintWriter out = response.getWriter();
          HttpSession session=request.getSession(false);
        String n=(String)session.getAttribute("uname");
        out.print("Hello "+n);
     out.close();
              }catch(Exception e){System.out.println(e);}
    }  }
```

web.xml

```
<web-app>
 <servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>
 <servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>
 <servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>
 <servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>
 </web-app>
```

Deployment Descriptor:

In a java web application a file named web.xml is known as deployment descriptor. It is a xml file and is the root element for it. When a request comes web server uses web.xml file to map the URL of the request to the specific code that handle the request.

Sample code of web.xml file:

```
<web-app>

 <servlet>
  <servlet-name>servletName</servlet-name>
  <servlet-class>servletClass</servlet-class>
 </servlet>

 <servlet-mapping>
  <servlet-name>servletName</servlet-name>
  <url-pattern>*.*</url-pattern>
 </servlet-mapping>

</web-app>
```

How web.xml works:

When a request comes it is matched with url pattern in servlet mapping attribute. In the above example all urls mapped with the servlet. You can specify a url pattern according to your need. When url matched with url pattern web server try to find the servlet name in servlet attributes same as in servlet mapping attribute. When match found control is goes to the associated servlet class.

Servlet "Hello World" example by extending HttpServlet class.

HelloWorld.java

```
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;

/**
 * This servlet program is used to print "Hello World"
 * on client browser using HttpServlet class.
 * @author java tutorial point
 */
public class HelloWorld extends HttpServlet {
    private static final long serialVersionUID = 1L;

    //no-argument constructor.
    public HelloWorld() {

    }

    protected void doGet(HttpServletRequest request,
HttpServletResponse
             response) throws ServletException, IOException {
 response.setContentType("text/html");
 PrintWriter out = response.getWriter();
```

```
        out.println("
```

Hello World using HttpServlet class.

```
");
        out.close();
    }
}
```

web.xml

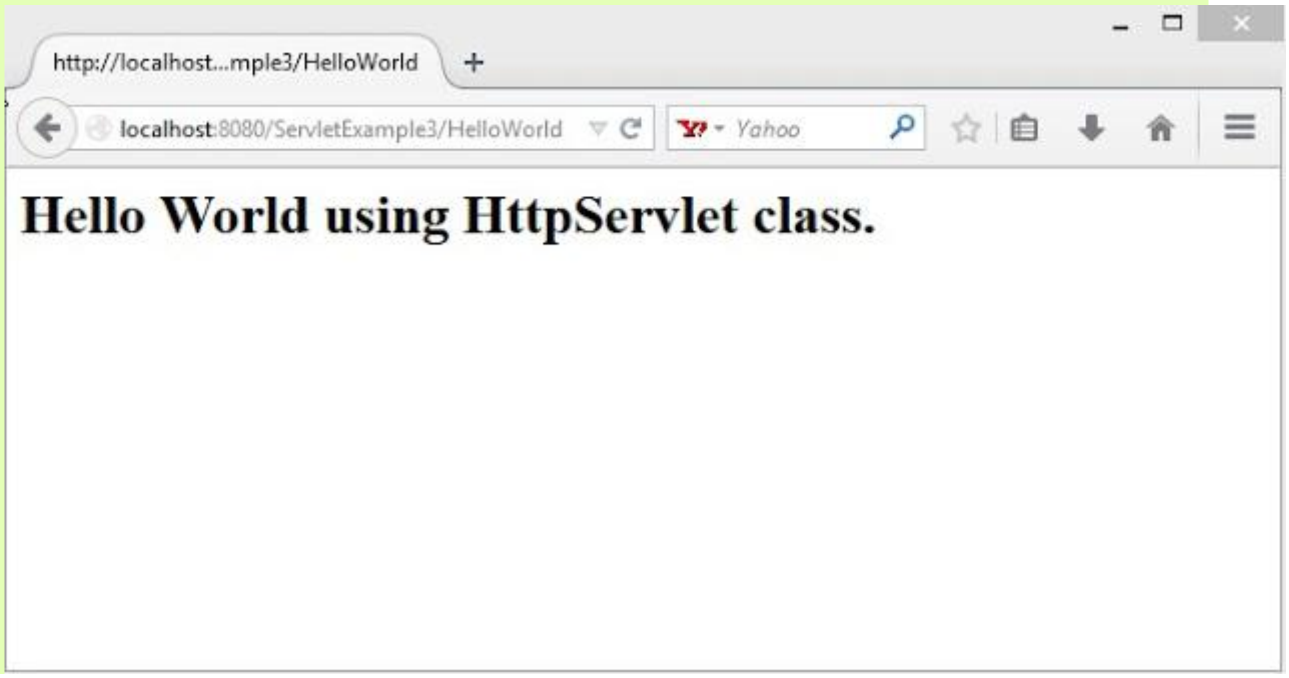
```
xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

 <servlet>
  <servlet-name>HelloWorld</servlet-name>
  <servlet-class>
    com.javawithease.business.HelloWorld
  </servlet-class>
 </servlet>

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

</web-app>
```

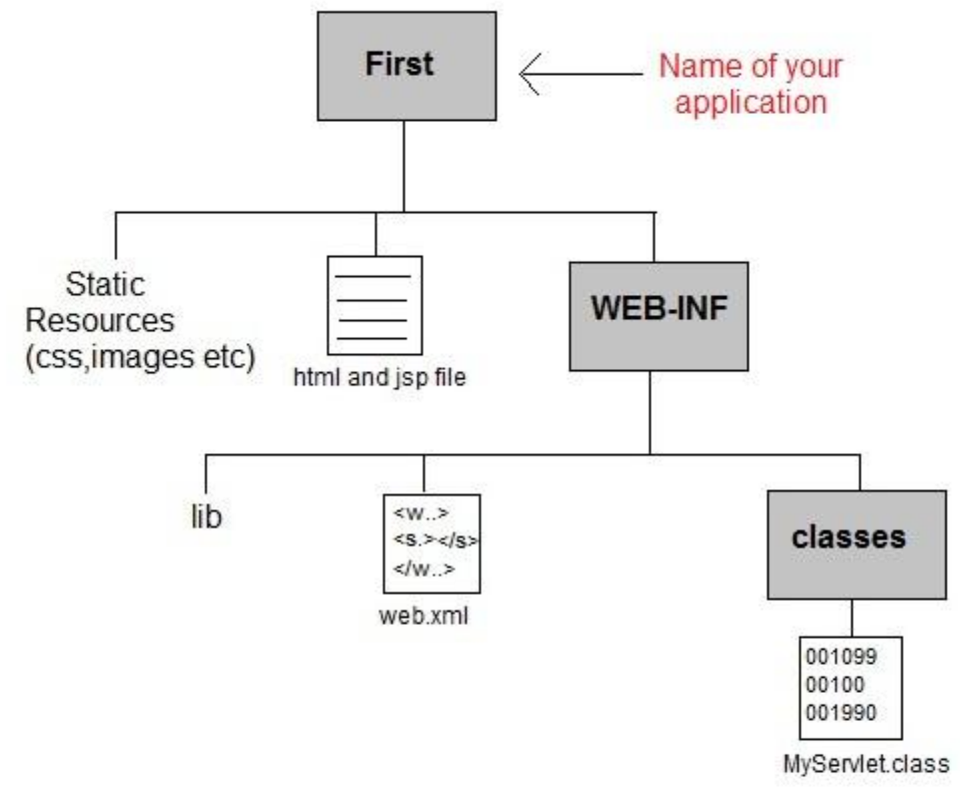
Output:


http://localhost...mple3/HelloWorld
localhost:8080/ServletExample3/HelloWorld
Yahoo

Hello World using HttpServlet class.

Creating the Directory Structure

Sun Microsystem defines a unique directory structure that must be followed to create a servlet application.



Create the above directory structure inside **Apache-Tomcat\webapps** directory. All HTML, static files(images, css etc) are kept directly under **Web application** folder. While all the Servlet classes are kept inside `classes` folder.

The `web.xml` (deployement descriptor) file is kept under `WEB-INF` folder.

servlet API

http://ecomputernotes.com/servlet/intro/servlet-api

SERVLETS - LIFE CYCLE

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet.

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

Now let us discuss the life cycle methods in detail.

The init Method

The init method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this –

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

The service Method

The service method is the main method to perform the actual task. The servlet container $i.e. webserver$ calls the service method to handle requests coming from the client$browsers$ and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service method checks the HTTP request type $GET, POST, PUT, DELETE, etc.$ and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Here is the signature of this method –

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

The service method is called by the container and service method invokes doGet, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service method but you override either doGet or doPost depending on what type of request you receive from the client.

The doGet and doPost are most frequently used methods with in each service request. Here is the signature of these two methods.

The doGet Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet method.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   // Servlet code
}
```

The doPost Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost method.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   // Servlet code
}
```

The destroy Method

The destroy method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

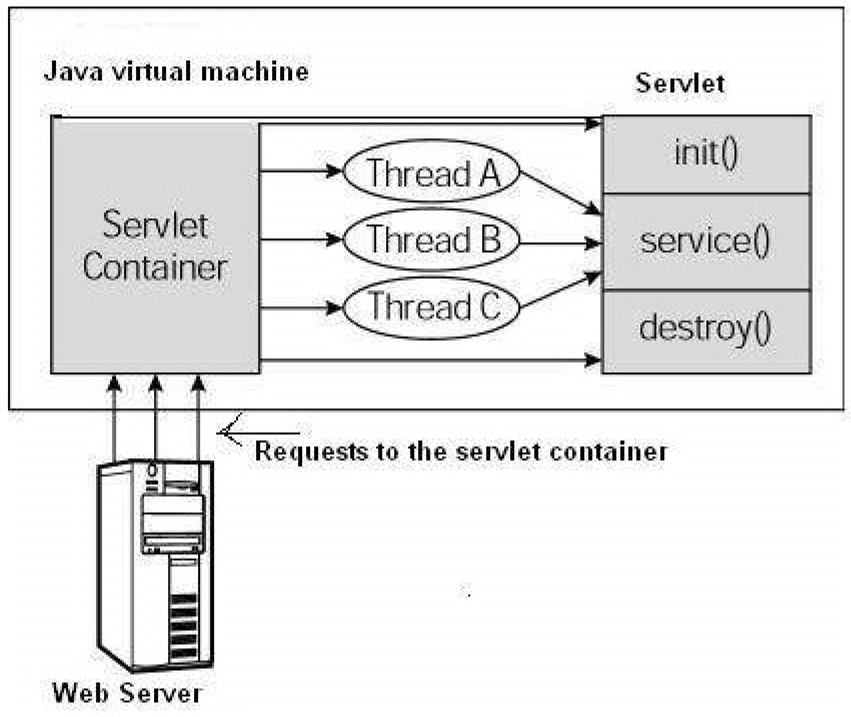
After the destroy method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this –

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

Architecture Diagram

The following figure depicts a typical 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.


Java virtual machine
Servlet
Servlet Container
Thread A
Thread B
Thread C
init()
service()
destroy()
Requests to the servlet container
Web Server

SERVLETS - FORM DATA

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your backend program. The browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

GET Method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the **?** *questionmark* symbol as follows –

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

The GET method is the default method to pass information from browser to web server and it produces a long string that appears in your browser's Location:box. Never use the GET method if you have password or other sensitive information to pass to the server. The GET method has size limitation: only 1024 characters can be used in a request string.

This information is passed using QUERY_STRING header and will be accessible through QUERY_STRING environment variable and Servlet handles this type of requests using **doGet** method.

POST Method

A generally more reliable method of passing information to a backend program is the POST method. This packages the information in exactly the same way as GET method, but instead of sending it as a text string after a ? *questionmark* in the URL it sends it as a separate message. This message comes to the backend program in the form of the standard input which you can parse and use for your processing. Servlet handles this type of requests using **doPost** method.

Reading Form Data using Servlet

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

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

GET Method Example using URL

Here is a simple URL which will pass two values to HelloForm program using GET method.

http://localhost:8080/HelloForm?first_name = ZARA&last_name = ALI

Given below is the **HelloForm.java** servlet program to handle input given by web browser. We are going to use **getParameter** method which makes it very easy to access passed information –

```
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Extend HttpServlet class
public class HelloForm extends HttpServlet {

   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      
      // Set response content type
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Using GET Method to Read Form Data";
      String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";
         
      out.println(docType +
         "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +
            "<body bgcolor = \"#f0f0f0\">\n" +
               "<h1 align = \"center\">" + title + "</h1>\n" +
               "<ul>\n" +
                  "  <li><b>First Name</b>: "
                  + request.getParameter("first_name") + "\n" +
                  "  <li><b>Last Name</b>: "
                  + request.getParameter("last_name") + "\n" +
               "</ul>\n" +
            "</body>" +
         "</html>"
      );
   }
}
```

Assuming your environment is set up properly, compile HelloForm.java as follows –

```
$ javac HelloForm.java
```

If everything goes fine, above compilation would produce **HelloForm.class** file. Next you would have to copy this class file in <Tomcat-installationdirectory>/webapps/ROOT/WEB-INF/classes and create following entries in **web.xml** file located in <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/

```
<servlet>
   <servlet-name>HelloForm</servlet-name>
   <servlet-class>HelloForm</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>HelloForm</servlet-name>
   <url-pattern>/HelloForm</url-pattern>
</servlet-mapping>
```

Now type *http://localhost:8080/HelloForm?first_name=ZARA&last_name=ALI* in your browser's Location:box and make sure you already started tomcat server, before firing above command in the browser. This would generate following result –

```
USING GET METHOD TO READ FORM DATA

• First Name: ZARA

• Last Name: ALI
```

GET Method Example Using Form

Here is a simple example which passes two values using HTML FORM and submit button. We are going to use same Servlet HelloForm to handle this input.

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

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

First Name: Last Name:

Try to enter First Name and Last Name and then click submit button to see the result on your local machine where tomcat is running. Based on the input provided, it will generate similar result as mentioned in the above example.

POST Method Example Using Form

Let us do little modification in the above servlet, so that it can handle GET as well as POST methods. Below is **HelloForm.java** servlet program to handle input given by web browser using GET or POST methods.

```java
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Extend HttpServlet class
```

```
public class HelloForm extends HttpServlet {

   // Method to handle GET method request.
   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {

      // Set response content type
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Using GET Method to Read Form Data";
      String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " +
         "transitional//en\">\n";

      out.println(docType +
         "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +
            "<body bgcolor = \"#f0f0f0\">\n" +
               "<h1 align = \"center\">" + title + "</h1>\n" +
               "<ul>\n" +
                  "  <li><b>First Name</b>: "
                  + request.getParameter("first_name") + "\n" +
                  "  <li><b>Last Name</b>: "
                  + request.getParameter("last_name") + "\n" +
               "</ul>\n" +
            "</body>"
         "</html>"
      );
   }

   // Method to handle POST method request.
   public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {

      doGet(request, response);
   }
}
```

Now compile and deploy the above Servlet and test it using Hello.htm with the POST method as follows –

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

Here is the actual output of the above form, Try to enter First and Last Name and then click submit button to see the result on your local machine where tomcat is running.

First Name: Last Name:

Based on the input provided, it would generate similar result as mentioned in the above examples.

Passing Checkbox Data to Servlet Program

Checkboxes are used when more than one option is required to be selected.

Here is example HTML code, CheckBox.htm, for a form with two checkboxes

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

The result of this code is the following form

Maths Physics Chemistry

Given below is the CheckBox.java servlet program to handle input given by web browser for checkbox button.

```
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Extend HttpServlet class
public class CheckBox extends HttpServlet {
 
   // Method to handle GET method request.
   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      
      // Set response content type
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Reading Checkbox Data";
      String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";

      out.println(docType +
         "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +
            "<body bgcolor = \"#f0f0f0\">\n" +
               "<h1 align = \"center\">" + title + "</h1>\n" +
               "<ul>\n" +
                  "  <li><b>Maths Flag : </b>: "
                  + request.getParameter("maths") + "\n" +
                  "  <li><b>Physics Flag: </b>: "
                  + request.getParameter("physics") + "\n" +
                  "  <li><b>Chemistry Flag: </b>: "
                  + request.getParameter("chemistry") + "\n" +
               "</ul>\n" +
            "</body>"
         "</html>"
      );
   }

   // Method to handle POST method request.
   public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {

      doGet(request, response);
   }
}
```

For the above example, it would display following result –

```
                    READING CHECKBOX DATA

   • Maths Flag : : on

   • Physics Flag: : null

   • Chemistry Flag: : on
```

Reading All Form Parameters

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

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

```
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// Extend HttpServlet class
public class ReadParams extends HttpServlet {

   // Method to handle GET method request.
   public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {

      // Set response content type
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Reading All Form Parameters";
      String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";

      out.println(docType +
         "<html>\n" +
         "<head><title>" + title + "</title></head>\n" +
         "<body bgcolor = \"#f0f0f0\">\n" +
         "<h1 align = \"center\">" + title + "</h1>\n" +
         "<table width = \"100%\" border = \"1\" align = \"center\">\n" +
         "<tr bgcolor = \"#949494\">\n" +
            "<th>Param Name</th>"
            "<th>Param Value(s)</th>\n"+
         "</tr>\n"
      );

      Enumeration paramNames = request.getParameterNames();

      while(paramNames.hasMoreElements()) {
         String paramName = (String)paramNames.nextElement();
         out.print("<tr><td>" + paramName + "</td>\n<td>");
         String[] paramValues = request.getParameterValues(paramName);

         // Read single valued data
         if (paramValues.length == 1) {
            String paramValue = paramValues[0];
            if (paramValue.length() == 0)
               out.println("<i>No Value</i>");
               else
               out.println(paramValue);
         } else {
            // Read multiple valued data
            out.println("<ul>");

            for(int i = 0; i < paramValues.length; i++) {
               out.println("<li>" + paramValues[i]);
            }
            out.println("</ul>");
         }
      }
      out.println("</tr>\n</table>\n</body></html>");
   }

   // Method to handle POST method request.
   public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {

      doGet(request, response);
   }
}
```

Now, try the above servlet with the following form –

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

Now calling servlet using the above form would generate the following result –

READING ALL FORM PARAMETERS

Param Name	Param Value(s)
maths	on
chemistry	on

You can try the above servlet to read any other form's data having other objects like text box, radio button or drop down box etc.

HTTP - REQUESTS

An HTTP client sends an HTTP request to a server in the form of a request message which includes following format:

```
• A Request-line

• Zero or more header (General|Request|Entity) fields followed by CRLF

• An empty line (i.e., a line with nothing preceding the CRLF)
  indicating the end of the header fields

• Optionally a message-body
```

The following sections explain each of the entities used in an HTTP request message.

Request-Line

The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by space SP characters.

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

Let's discuss each of the parts mentioned in the Request-Line.

Request Method

The request **method** indicates the method to be performed on the resource identified by the given **Request-URI**. The method is case-sensitive and should always be mentioned in uppercase. The following table lists all the supported methods in HTTP/1.1.

S.N.	Method and Description
1	**GET** The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.
2	**HEAD** Same as GET, but it transfers the status line and the header section only.
3	**POST** A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.
4	**PUT**

	Replaces all the current representations of the target resource with the uploaded content.
5	**DELETE** Removes all the current representations of the target resource given by URI.
6	**CONNECT** Establishes a tunnel to the server identified by a given URI.
7	**OPTIONS** Describe the communication options for the target resource.
8	**TRACE** Performs a message loop back test along with the path to the target resource.

Request-URI

The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request. Following are the most commonly used forms to specify an URI:

```
Request-URI = "*" | absoluteURI | abs_path | authority
```

S.N.	Method and Description
1	The asterisk ***** is used when an HTTP request does not apply to a particular resource, but to the server itself, and is only allowed when the method used does not necessarily apply to a resource. For example: **OPTIONS * HTTP/1.1**
2	The **absoluteURI** is used when an HTTP request is being made to a proxy. The proxy is requested to forward the request or service from a valid cache, and return the response. For example: **GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1**
3	The most common form of Request-URI is that used to identify a resource on an origin server or gateway. For example, a client wishing to retrieve a resource directly from the origin server would create a TCP connection to port 80 of the host "www.w3.org" and send the following lines: **GET /pub/WWW/TheProject.html HTTP/1.1** **Host: www.w3.org** Note that the absolute path cannot be empty; if none is present in the original URI, it MUST be given as "/" *theserverroot*.

Request Header Fields

We will study General-header and Entity-header in a separate chapter when we will learn HTTP header fields. For now, let's check what Request header fields are.

The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers.Here is a list of some important Request-header fields that can be used based on the requirement:

- Accept-Charset
- Accept-Encoding
- Accept-Language
- Authorization
- Expect
- From
- Host
- If-Match
- If-Modified-Since
- If-None-Match
- If-Range
- If-Unmodified-Since
- Max-Forwards
- Proxy-Authorization
- Range
- Referer
- TE
- User-Agent

You can introduce your custom fields in case you are going to write your own custom Client and Web Server.

Examples of Request Message

Now let's put it all together to form an HTTP request to fetch **hello.htm** page from the web server running on tutorialspoint.com

```
GET /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Here we are not sending any request data to the server because we are fetching a plain HTML page from the server. Connection is a general-header, and the rest of the headers are request headers. The following example shows how to send form data to the server using request message body:

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
```

```
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

licenseID=string&content=string&/paramsXML=string
```

Here the given URL */cgi-bin/process.cgi* will be used to process the passed data and accordingly, a response will be returned. Here **content-type** tells the server that the passed data is a simple web form data and **length** will be the actual length of the data put in the message body. The following example shows how you can pass plain XML to your web server:

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: text/xml; charset=utf-8
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://clearforest.com/">string</string>
```

HTTP - RESPONSES

After receiving and interpreting a request message, a server responds with an HTTP response message:

```
• A Status-line

• Zero or more header (General|Response|Entity) fields followed by CRLF

• An empty line (i.e., a line with nothing preceding the CRLF)
  indicating the end of the header fields

• Optionally a message-body
```

The following sections explain each of the entities used in an HTTP response message.

Message Status-Line

A Status-Line consists of the protocol version followed by a numeric status code and its associated textual phrase. The elements are separated by space SP characters.

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

HTTP Version

A server supporting HTTP version 1.1 will return the following version information:

```
HTTP-Version = HTTP/1.1
```

Status Code

The Status-Code element is a 3-digit integer where first digit of the Status-Code defines the class of response and the last two digits do not have any categorization role. There are 5 values for the first digit:

S.N.	Code and Description
1	**1xx: Informational** It means the request was received and the process is continuing.
2	**2xx: Success** It means the action was successfully received, understood, and accepted.
3	**3xx: Redirection** It means further action must be taken in order to complete the request.
4	**4xx: Client Error**

It means the request contains incorrect syntax or cannot be fulfilled.

5 **5xx: Server Error**

It means the server failed to fulfill an apparently valid request.

HTTP status codes are extensible and HTTP applications are not required to understand the meaning of all registered status codes. A list of all the status codes has been given in a separate chapter for your reference.

Response Header Fields

We will study General-header and Entity-header in a separate chapter when we will learn HTTP header fields. For now, let's check what Response header fields are.

The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status- Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

- Accept-Ranges
- Age
- ETag
- Location
- Proxy-Authenticate
- Retry-After
- Server
- Vary
- WWW-Authenticate

You can introduce your custom fields in case you are going to write your own custom Web Client and Server.

Examples of Response Message

Now let's put it all together to form an HTTP response for a request to fetch the **hello.htm** page from the web server running on tutorialspoint.com

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
```

```
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

The following example shows an HTTP response message displaying error condition when the web server could not find the requested page:

```
HTTP/1.1 404 Not Found
```

```
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 230
Connection: Closed
Content-Type: text/html; charset=iso-8859-1
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
   <title>404 Not Found</title>
</head>
<body>
   <h1>Not Found</h1>
   <p>The requested URL /t.html was not found on this server.</p>
</body>
</html>
```

Following is an example of HTTP response message showing error condition when the web server encountered a wrong HTTP version in the given HTTP request:

```
HTTP/1.1 400 Bad Request
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 230
Content-Type: text/html; charset=iso-8859-1
Connection: Closed
```

```

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
   <title>400 Bad Request</title>
</head>
<body>
   <h1>Bad Request</h1>
   <p>Your browser sent a request that this server could not understand.</p>
   <p>The request line contained invalid characters following the protocol string.</p>
</body>
</html>
```

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.

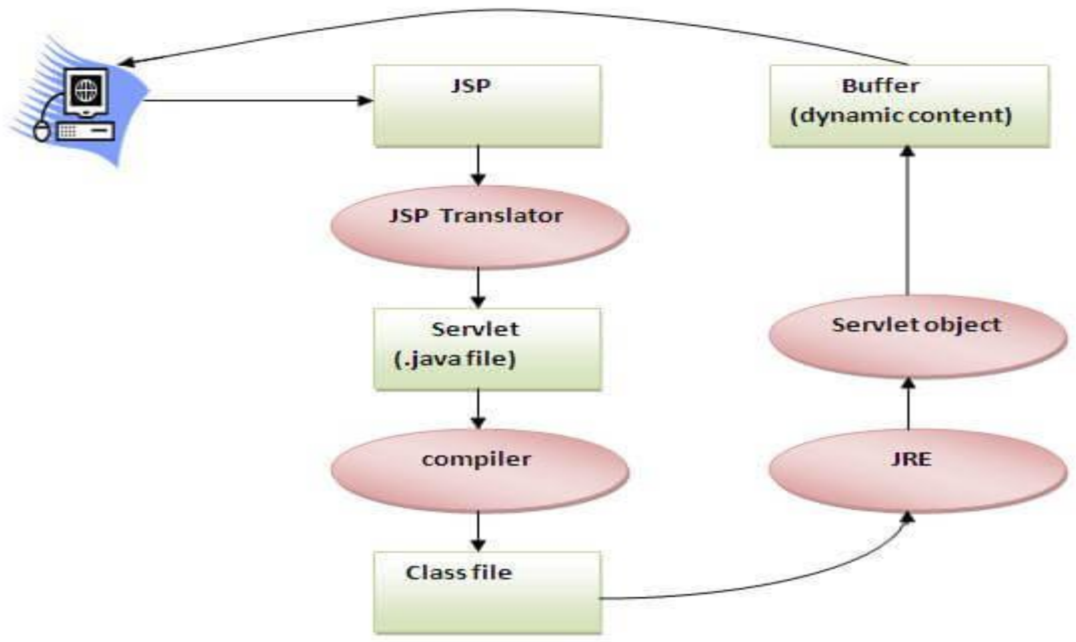
4) Less code than Servlet

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.

The Lifecycle of a JSP Page

The JSP pages follow these phases:

- Translation of JSP Page
- Compilation of JSP Page
- Classloading (the classloader loads class file)
- Instantiation (Object of the Generated Servlet is created).
- Initialization (the container invokes jspInit() method).
- Request processing (the container invokes _jspService() method).
- Destroy (the container invokes jspDestroy() method).



As depicted in the above diagram, JSP page is translated into Servlet by the help of JSP translator. The JSP translator is a part of the web server which is responsible for translating the JSP page into Servlet. After that, Servlet page is compiled by the compiler and gets converted into the class file. Moreover, all the processes that happen in Servlet are performed on JSP later like initialization, committing response to the browser and destroy

Creating a simple JSP Page

To create the first JSP page, write some HTML code as given below, and save it by .jsp extension. We have saved this file as index.jsp. Put it in a folder and paste the folder in the web-apps directory in apache tomcat to run the JSP page.

index.jsp

Let's see the simple example of JSP where we are using the scriptlet tag to put Java code in the JSP page. We will learn scriptlet tag later.

```
<html>
<body>
<% out.print(2*5); %>
</body>
</html>
```

It will print **10** on the browser.

How to run a simple JSP Page?

Follow the following steps to execute this JSP page:

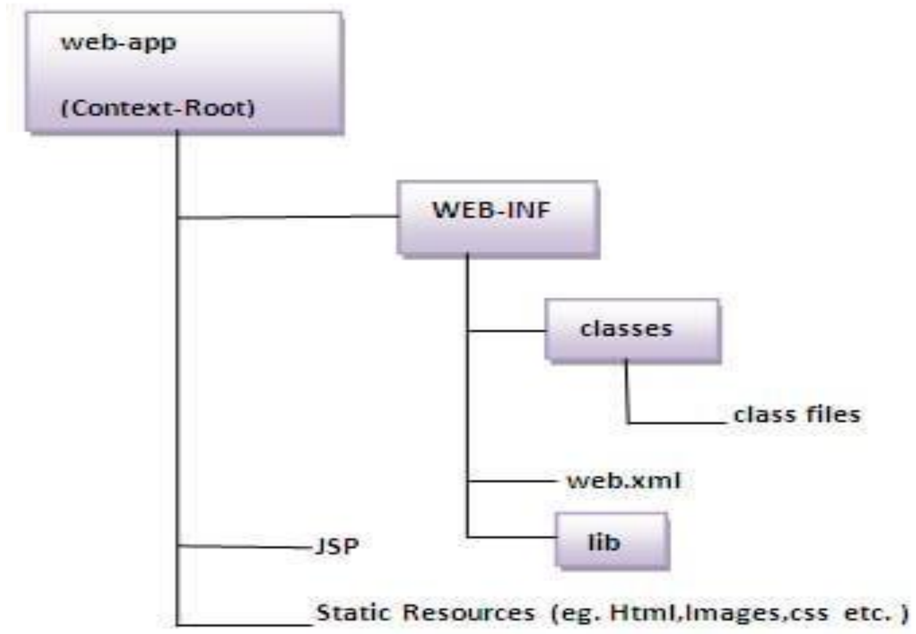
- Start the server
- Put the JSP file in a folder and deploy on the server
- Visit the browser by the URL http://localhost:portno/contextRoot/jspfile, for example, http://localhost:8888/myapplication/index.jsp

Do I need to follow the directory structure to run a simple JSP?

No, there is no need of directory structure if you don't have class files or TLD files. For example, put JSP files in a folder directly and deploy that folder. It will be running fine. However, if you are using Bean class, Servlet or TLD file, the directory structure is required.

The Directory structure of JSP

The directory structure of JSP page is same as Servlet. We contain the JSP page outside the WEB-INF folder or in any directory.



The JSP API

The JSP API consists of two packages:

1. javax.servlet.jsp
2. javax.servlet.jsp.tagext

javax.servlet.jsp package

The javax.servlet.jsp package has two interfaces and classes.The two interfaces are as follows:

1. JspPage
2. HttpJspPage

The classes are as follows:

- JspWriter
- PageContext
- JspFactory
- JspEngineInfo
- JspException
- JspError

The JspPage interface

According to the JSP specification, all the generated servlet classes must implement the JspPage interface. It extends the Servlet interface. It provides two life cycle methods.

Methods of JspPage interface

1. **public void jspInit():** It is invoked only once during the life cycle of the JSP when JSP page is requested firstly. It is used to perform initialization. It is same as the init() method of Servlet interface.
2. **public void jspDestroy():** It is invoked only once during the life cycle of the JSP before the JSP page is destroyed. It can be used to perform some clean up operation.

The HttpJspPage interface

The HttpJspPage interface provides the one life cycle method of JSP. It extends the JspPage interface.

Method of HttpJspPage interface:

1. **public void _jspService():** It is invoked each time when request for the JSP page comes to the container. It is used to process the request. The underscore _ signifies that you cannot override this method.

Introduction:

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.

A JSP page consists of HTML tags and JSP tags. The JSP pages are easier to maintain than Servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tags, etc.

Advantages:

1. Extension to Servlet
2. Easy to maintain
3. Less code than Servlet
4. Less Development: No need to recompile and redeploy

JSP directives

The **jsp directives** are messages that tells the web container how to translate a JSP page into the corresponding servlet.

There are three types of directives:

- page directive
- include directive
- taglib directive

Syntax of JSP Directive

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

JSP page directive

The page directive defines attributes that apply to an entire JSP page.

Syntax of JSP page directive

```
1. <%@ page attribute="value" %>
```

Attributes of JSP page directive

- import
- contentType
- extends
- info
- buffer
- language
- isELIgnored
- isThreadSafe
- autoFlush
- session
- pageEncoding
- errorPage
- isErrorPage

1)import

The import attribute is used to import class,interface or all the members of a package.It is similar to imp
class or interface.

Example of import attribute

```
<html>
<body>

<%@ page import="java.util.Date" %>
Today is: <%= new Date() %>

</body>
</html>
```

2)contentType

The contentType attribute defines the MIME(Multipurpose Internet Mail Extension) type of the HTTP response.The default value is "text/html;charset=ISO-8859-1".

Example of contentType attribute

```
<html>
<body>

<%@ page contentType=application/msword %>
Today is: <%= new java.util.Date() %>

</body>
</html>
```

3)extends

The extends attribute defines the parent class that will be inherited by the generated servlet.It is rarely used.

4)info

This attribute simply sets the information of the JSP page which is retrieved later by using getServletInfo() method of Servlet interface.

Example of info attribute

```
<html>
<body>

<%@ page info="composed by Sonoo Jaiswal" %>
Today is: <%= new java.util.Date() %>

</body>
</html>
```

The web container will create a method getServletInfo() in the resulting servlet.For example:

```
public String getServletInfo() {
  return "composed by Sonoo Jaiswal";
}
```

---

5)buffer

The buffer attribute sets the buffer size in kilobytes to handle output generated by the JSP page.The default size of the buffer is 8Kb.

Example of buffer attribute

```
<html>
<body>

<%@ page buffer="16kb" %>
Today is: <%= new java.util.Date() %>

</body>
</html>
```

---

6)language

The language attribute specifies the scripting language used in the JSP page. The default value is "java".

---

7)isELIgnored

We can ignore the Expression Language (EL) in jsp by the isELIgnored attribute. By default its value is f

Language is enabled by default. We see Expression Language later.

```
<%@ page isELIgnored="true" %>//Now EL will be ignored
```

8)isThreadSafe

Servlet and JSP both are multithreaded.If you want to control this behaviour of JSP page, you can use is
of page directive.The value of isThreadSafe value is true.If you make it false, the web container will seri
requests, i.e. it will wait until the JSP finishes responding to a request before passing another request to
value of isThreadSafe attribute like:

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

The web container in such a case, will generate the servlet as:

```
public class SimplePage_jsp extends HttpJspBase
  implements SingleThreadModel{
.......
}
```

9)errorPage

The errorPage attribute is used to define the error page, if exception occurs in the current page, it will be redirected to the error page.

Example of errorPage attribute

```
//index.jsp
<html>
<body>

<%@ page errorPage="myerrorpage.jsp" %>

 <%= 100/0 %>

</body>
</html>
```

10)isErrorPage

The isErrorPage attribute is used to declare that the current page is the error page.

Note: The exception object can only be used in the error page.

Example of isErrorPage attribute

```
//myerrorpage.jsp
<html>
<body>

<%@ page isErrorPage="true" %>

 Sorry an exception occured!<br/>
The exception is: <%= exception %>

</body>
</html>
```

TAGS in JSP:

Declaration tag:
Whenever we use any variables as a part of JSP we have to use those variables in the form of declaration tag i.e., declaration tag is used for declaring the variables in JSP page.

Syntax:

<%! Variable declaration or method definition %>

Example-1:

```
<%!    int a = 10, b = 30, c;%>
<%!    int a = 10, b = 30, c;%>
```

Expression tag:
Expression tags are used for writing the java valid expressions as a part of JSP page.

Syntax:

```
<%= java valid expression %>
```

Example-1:

```
<%! int a = 10, b = 20 %>
<%= a + b%>
```

Scriplet tag:
Scriplets are basically used to write a pure java code. Whatever the java code we write as a part of scriplet, that code will be available as a part of service () method of servlet.

Syntax:

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

Implicit Objects in JSP:

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

Following list is the nine Implicit Objects that JSP supports −

1. request:

This is the HttpServletRequest object associated with the request.

2. response:

This is the HttpServletResponse object associated with the response to the client.

3. out:

This is the PrintWriter object used to send output to the client.

4. session:

This is the HttpSession object associated with the request.

5. application:

This is the ServletContext object associated with the application context.

6. config:

This is the ServletConfig object associated with the page.

7. pageContext:

This encapsulates use of server-specific features like higher performance JspWriters.

8. page:

This is simply a synonym for this, and is used to call the methods defined by the translated servlet class.

9. exception:

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

JSP Methods:

JSP Declaration represents a global area for the whole JSP file where programmer can declare variables and methods that can be used throughout JSP code. That is, global variables and methods are declared in Declaration tag.

Example:

```
<HTML>
  <HEAD>
    <TITLE>Creating a Method in jsp</TITLE>
  </HEAD>
<BODY>
    <H1>Creating a Method in jsp</H1>
    <%!
    Int Double(int number)
    {
      return 2*number;
    }
    %>

    <%
    out.println("The Double of 3 is = " + Double(3));
    %>
  </BODY>
</HTML>
```

Control-Flow Statements:

You can use all the APIs and building blocks of Java in your JSP programming including decision-making statements, loops, etc.

```
<%! 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>
```

Loop Statements:

You can also use three basic types of looping blocks in Java: for, while, and do…while blocks in your JSP programming.

```
<%! int fontSize; %>
<html>
   <head><title>FOR LOOP Example</title></head>

   <body>
      <%for ( fontSize = 1; fontSize<= 3; fontSize++){ %>
      <font color = "green" size = "<%= fontSize %>">
        JSP Tutorial
    </font><br />
    <%}%>
  </body>
</html>
```

JSP request implicit object:

The JSP request is an implicit object of type HttpServletRequest i.e. created for each jsp request by the web container. It can be used to get request information such as parameter, header information, remote address, server name, server port, content type, character encoding etc.

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

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

Example of JSP request implicit object

index.html

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

welcome.jsp

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

JSP Session

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

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

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

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

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

<input type = "hidden" name = "sessionid" value = "12345">
This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or the POST data. Each time the web browser sends the request back, the session_id value can be used to keep the track of different web browsers.

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

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

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

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

The session Object

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

a one page request or
visit to a website or
store information about that user
By default, JSPs have session tracking enabled and a new HttpSession object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows −

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

Here is a summary of important methods available through the session object −

S.No.	Method & Description
1	**public Object getAttribute(String name)** This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	**public Enumeration getAttributeNames()** This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3	**public long getCreationTime()** This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.

4	**public String getId()** This method returns a string containing the unique identifier assigned to this session.
5	**public long getLastAccessedTime()** This method returns the last time the client sent a request associated with the this session, as the number of milliseconds since midnight January 1, 1970 GMT.
6	**public int getMaxInactiveInterval()** This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
7	**public void invalidate()** This method invalidates this session and unbinds any objects bound to it.
8	**public booleanisNew()** This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
9	**public void removeAttribute(String name)** This method removes the object bound with the specified name from this session.
10	**public void setAttribute(String name, Object value)** This method binds an object to this session, using the name specified.
11	**public void setMaxInactiveInterval(int interval)** This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

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

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

```
   // Get session creation time.
   Date createTime = new Date(session.getCreationTime());

   // Get last access time of this Webpage.
   Date lastAccessTime = new Date(session.getLastAccessedTime());

   String title = "Welcome Back to my website";
   Integer visitCount = new Integer(0);
   String visitCountKey = new String("visitCount");
   String userIDKey = new String("userID");
   String userID = new String("ABCD");

   // Check if this is new comer on your Webpage.
   if (session.isNew() ){
      title = "Welcome to my website";
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>
<head>
<title>Session Tracking</title>
</head>

<body>
<center>
<h1>Session Tracking</h1>
</center>

<table border = "1" align = "center">
<tr bgcolor = "#949494">
<th>Session info</th>
<th>Value</th>
</tr>
<tr>
```

```
<td>id</td>
<td><% out.print( session.getId()); %></td>
</tr>
<tr>
<td>Creation Time</td>
<td><% out.print(createTime); %></td>
</tr>
<tr>
<td>Time of Last Access</td>
<td><% out.print(lastAccessTime); %></td>
</tr>
<tr>
<td>User ID</td>
<td><% out.print(userID); %></td>
</tr>
<tr>
<td>Number of visits</td>
<td><% out.print(visitCount); %></td>
</tr>
</table>

</body>
</html>
```

JSP Cookies:

JSP Cookies – Cookies can defined as a small file that will be stored in a browser, it is utilized information tracing purpose. Already Splessons have discussed the cookies concept in servlet technology Servlet Cookies. Following is the list of important useful methods associated with the Cookie object which you can use while manipulating cookies in JSP.

- public void setDomainStringpattern
- public String getDomain
- public String getName
- public String getValue
- public String getPath
- public String getComment

Following is the example which describes more about the cookies. Following is the code to set the cookies.

index.jsp

```
<html>
<body bgcolor="skyblue">
<form action="main.jsp" method="GET">
<center><imgsrc="E:/splessons.png"></br></br>
First Name: <input type="text" name="first_name"></br>
<br />
Last Name: <input type="text" name="last_name"/></br></br>
<input type="submit" value="Submit" /></center>
</form>
</body>
</html>
```

Here just created two forms they are First Name, Last Name and also created **Submit** button. The **GET**method is the default method to pass data from client to web server and it delivers a long string that shows up in the browser’s Location. Never utilize the GET method in the event that you have secret key or other touchy data to go to the server. The GET method has size constraint: just 1024 characters can be in a solicitation string.

main.jsp

```
<%
// Create cookies for first and last names.
Cookie firstName = new Cookie("first_name",
request.getParameter("first_name"));
Cookie lastName = new Cookie("last_name",
request.getParameter("last_name"));
// Set expiry date after 24 Hrs for both the cookies.
firstName.setMaxAge(60*60*24);
lastName.setMaxAge(60*60*24);
// Add both the cookies in the response header.
response.addCookie( firstName );
response.addCookie( lastName );
%>
```

```
<html>
<head>
<title>Setting Cookies</title>
</head>
<body>
<center>
<h1>Setting Cookies</h1>
</center>
<ul>
<li><p><b>First Name:</b>
<%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last Name:</b>
<%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>
```

The **request.getParameter()** is used to retrieve the details from the static page that id **HTML** page.