

## UNIT - II

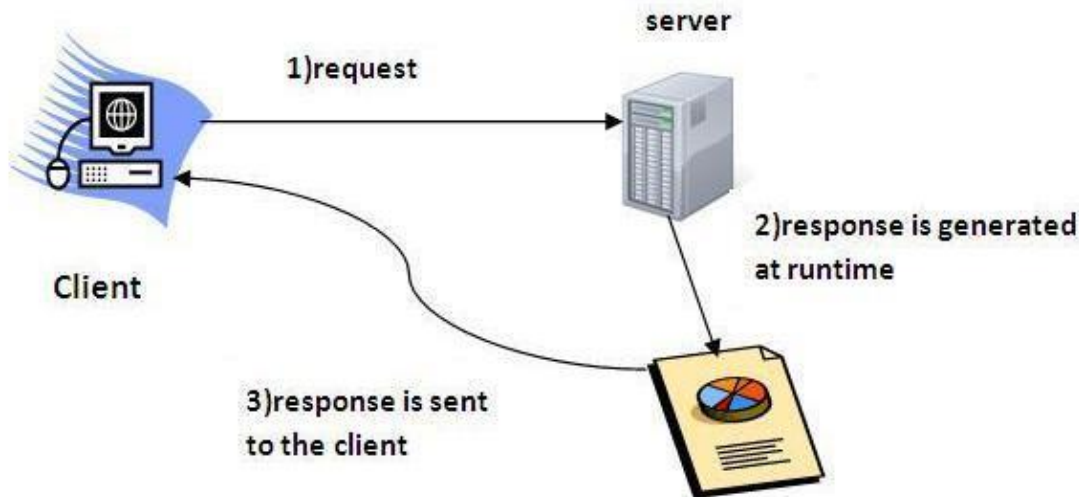
Servlet: Servlet structure, Life Cycle of a Servlet, Using Tomcat for Servlet Development, The Servlet API, Handling Client Request: Form data, Handling client HTTP request and server HTTP Response, HTTP status codes, Handling Cookies, Session tracking, Database Access

## UNIT-II SERVLETS

### 2.1 Introduction to Servlets

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

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



**Figure 2.1: Servlet's basic components**

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.

**Servlet 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.

Java servlets have been created and compiled just like any other Java class. After you install the servlet packages and add them to your computer's Classpath, you can compile servlets with the JDK's Java compiler or any other current compiler.

## Web Application

A web application is an application accessible from the web. A web application is composed of web components like Servlet, JSP, Filter etc. and other components such as HTML. The web components typically execute in Web Server and respond to HTTP request.

## Common Gateway Interface (CGI)

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.

## Disadvantages of CGI

There are many problems in CGI technology:

- ☐ If number of clients increases, it takes more time for sending response.
- ☐ For each request, it starts a process and Web server is limited to start processes.
- ☐ It uses platform dependent language e.g. C, C++, Perl.

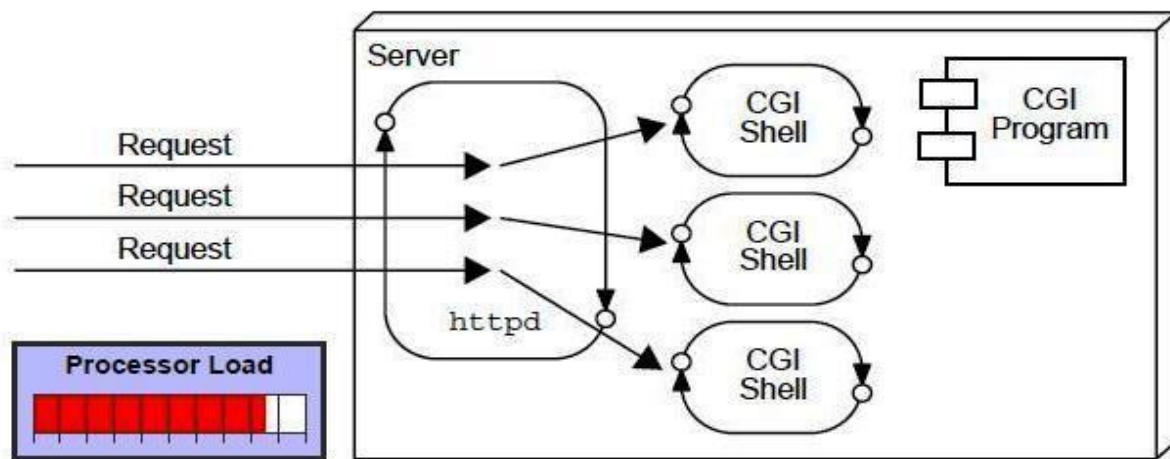
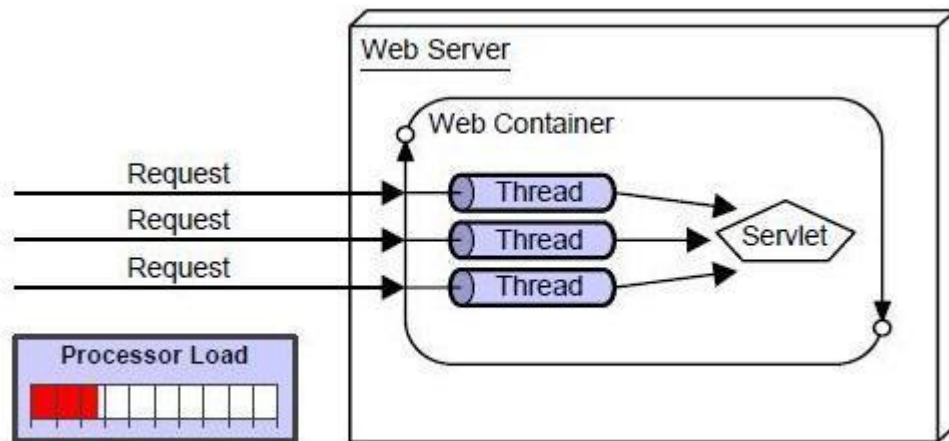


Figure 2.2: Common gateway Interface

## Advantages of Servlets

Java Servlets often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). Threads have a lot of benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. 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.



**Figure 2.3: Servlet Web Container**

## 2.2 Life Cycle of a Servlet

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 details.

### The `init()` method :

The `init` method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request. 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. web server) calls the `service()` method to handle requests coming from the client (browsers) and to write the formatted response back to the client.

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

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 within 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...
}
```

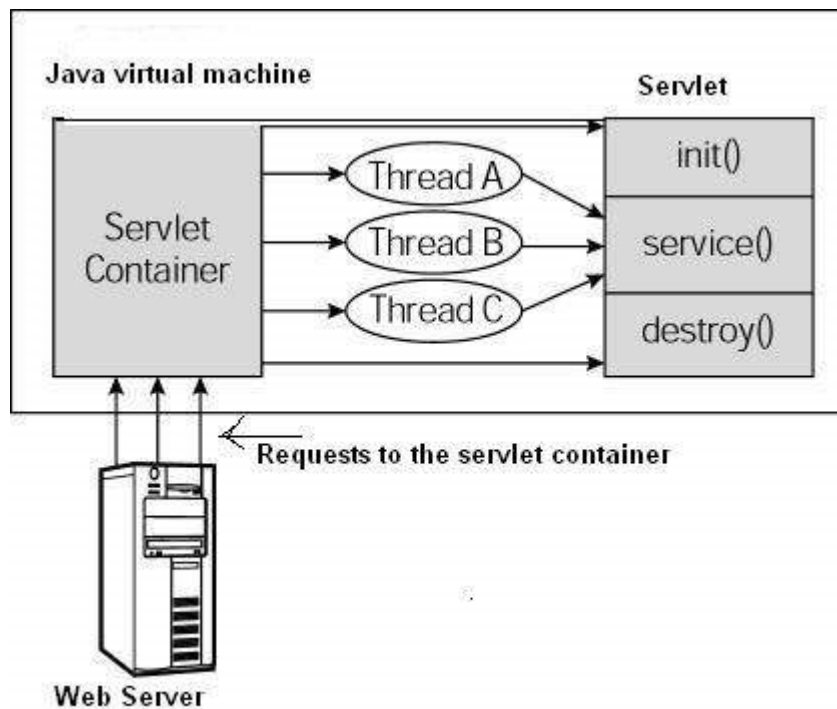
### **Architecture Diagram:**

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.



**Figure 2.4: Servlet Life cycle**

### 2.3 Using Tomcat for Servlet Development

To create a Servlet application you need to follow the below mentioned steps. These steps are common for all the Web server. In our example we are using Apache Tomcat server. Apache Tomcat is an open source web server for testing servlets and JSP technology. Download latest version of Tomcat Server and install it on your machine.

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

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

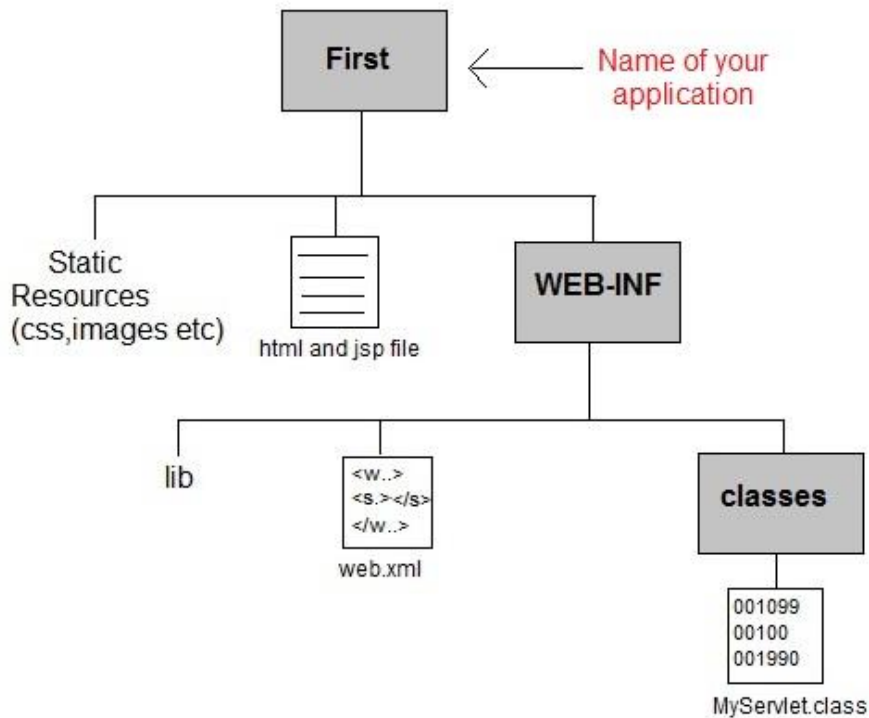
All these 5 steps are explained in details below; let's create our first Servlet Application.

#### **Create directory structure for your application.**

Sun Microsystems 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` (deployment descriptor) file is kept under `WEB-INF` folder.



**Figure 2.5: Servlet directory structure**

### Creating Servlet

There are three different ways to create a servlet.

- ☐ By implementing Servlet interface
- ☐ By extending GenericServlet class
- ☐ By extending HttpServlet class

But mostly a servlet is created by extending HttpServlet abstract class. As discussed earlier HttpServlet gives the definition of service() method of the Servlet interface. The servlet class that we will create should not override service() method. Our servlet class will override only doGet() or doPost() method.

When a request comes in for the servlet, the Web Container calls the servlet's service() method and depending on the type of request the service() method calls either the doGet() or doPost() method.

**Note:** By default a request is Get request.

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

```
public MyServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.println("<h1>Hello Readers</h1>");
        out.println("</body></html>");
    }
}
```



```
}  
}
```

Write above code in a notepad file and save it as MyServlet.java anywhere on your PC. Compile it(explained in next step) from there and paste the class file into WEB-INF/classes/ directory that you have to create inside Tomcat/webapps directory.

### Compiling a Servlet

To compile a Servlet a JAR file is required. Different servers require different JAR files. In Apache Tomcat server servlet-api.jar file is required to compile a servlet class.

Steps to compile a Servlet

- ☐ Set the Class Path.

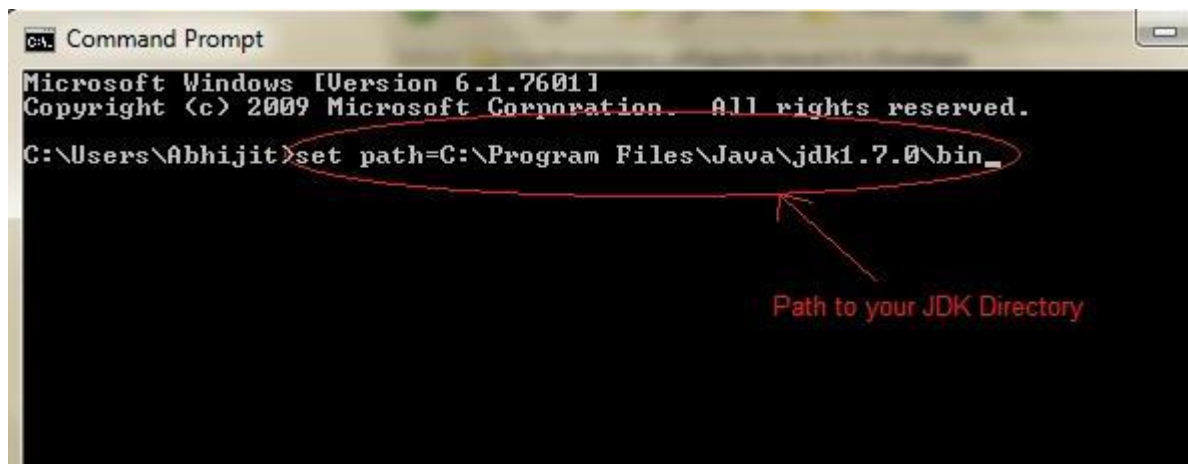


Figure 2.6: Setting Class Path

- ☐ Download **servlet-api.jar** file.
- ☐ Paste the servlet-api.jar file inside Java\jdk\jre\lib\ext directory.

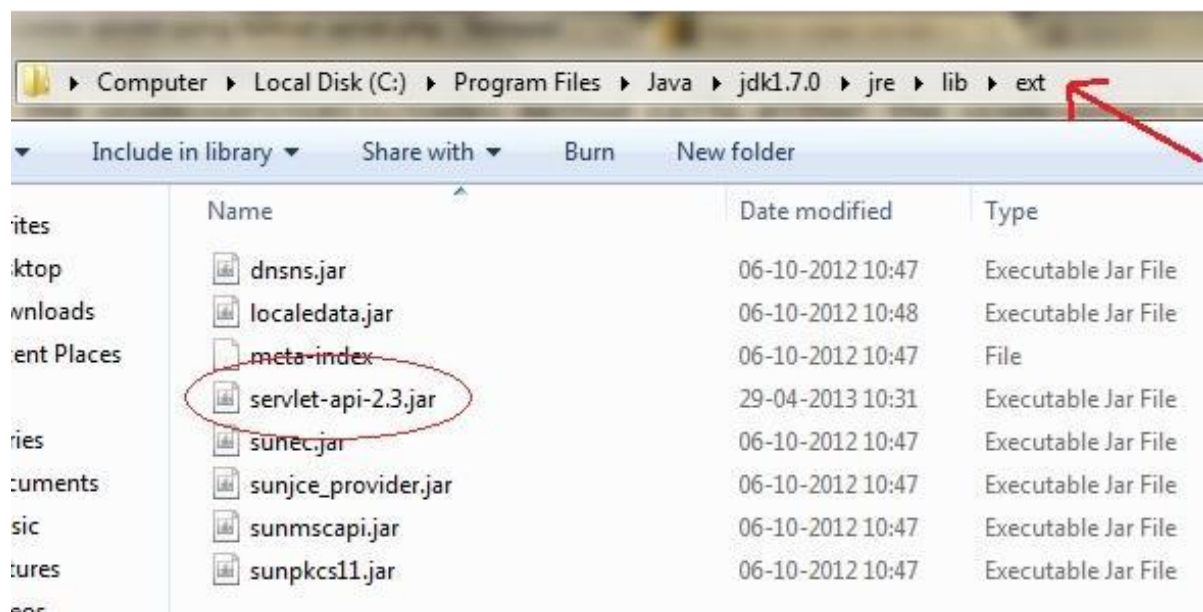
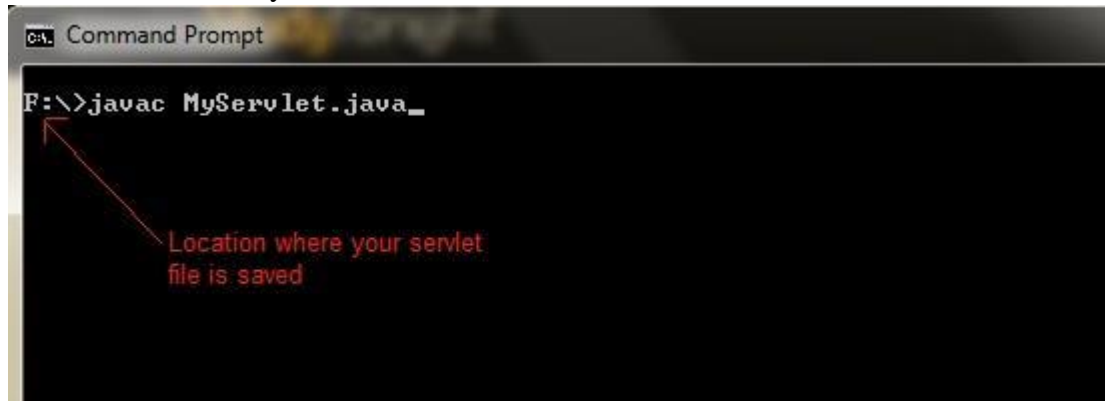


Figure 2.7: Adding Jar Files to Java Directory

- ☐ Compile the Servlet class.

**NOTE:** After compiling your Servlet class you will have to paste the class file into WEB-INF/classes/ directory.



**Figure 2.8: Compiling Servlet**

### Creating Deployment Descriptor

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

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

We will discuss about all these in details later. Now we will see how to create a simple web.xml file for our web application.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>MyServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```

Annotations in the image:

- First line of any xml document (points to the XML declaration line)
- root tag of wex.xml file. All other tag come inside it (points to the <web-app> tag)
- this tag maps internal name to fully qualified class name (points to the <servlet> tag)
- Give a internal name to your servlet (points to the <servlet-name>hello</servlet-name> line)
- servlet class that you have created (points to the <servlet-class>MyServlet</servlet-class> line)
- this tag maps internal name to public URL name (points to the <servlet-mapping> tag)
- URL name. This is what the user will see to get to the servlet. (points to the <url-pattern>/hello</url-pattern> line)

**Figure 2.9 Web-XML file**

---

### Starting Tomcat Server for the first time

If you are starting Tomcat Server for the first time you need to set JAVA\_HOME in the Enviroment variable. The following steps will show you how to set it.

- ☐ Right Click on My Computer, go to Properites
- ☐ Go to Advanced Tab and Click on Enviroment Variables... button.
- ☐ Click on New button, and enter JAVA\_HOME inside Variable name text field and path of JDK inside Variable value text field. Click OK to save.

### Run Servlet Application

Open Browser and type **http:localhost:8080/First/hello**



**Figure 2.10: Running Servlet Application in the browser**

### 16.4 The Servlet API

Servlet API consists of two important packages that encapsulate all the important classes and interface, namely:

javax.servlet  
javax.servlet.http

#### Some Important Classes and Interfaces of javax.servlet

INTERFACES	CLASSES
Servlet	ServletInputStream
ServletContext	ServletOutputStream
ServletConfig	ServletRequestWrapper
ServletRequest	ServletResponseWrapper
ServletResponse	ServletRequestEvent
ServletContextListener	ServletContextEvent
RequestDispatcher	ServletRequestAttributeEvent
SingleThreadModel	ServletContextAttributeEvent
Filter	ServletException
FilterConfig	UnavailableException
FilterChain	GenericServlet

ServletRequestListener

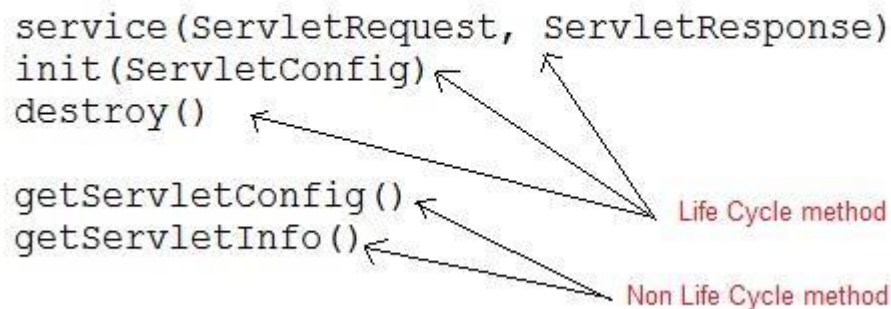
## Some Important Classes and Interface of javax.servlet.http

### CLASSES and INTERFACES

HttpServlet	HttpServletRequest
HttpServletResponse	HttpSessionAttributeListener
HttpSession	HttpSessionListener
Cookie	HttpSessionEvent

### Servlet Interface

Servlet Interface provides five methods. Out of these five methods, three methods are Servlet life cycle methods and rest two are non life cycle methods.



## 2.5 Handling HTTP request and Response

When a browser requests for a web page, it sends lot of information to the web server which cannot be read directly because this information travel as a part of header of HTTP request. You can check HTTP Protocol for more information on this.

Following is the important header information which comes from browser side and you would use very frequently in web programming:

Header	Description
Accept	This header specifies the MIME types that the browser or other clients can handle. Values of image/png or image/jpeg are the two most common possibilities.
Accept-Charset	This header specifies the character sets the browser can use to display the information. For example ISO-8859-1.
Accept-Encoding	This header specifies the types of encodings that the browser knows how to handle. Values of gzip or compress are the two most common possibilities.
Accept-Language	This header specifies the client's preferred languages in case the servlet can produce results in more than one language. For example en, en-us, ru, etc.
Authorization	This header is used by clients to identify themselves when accessing password-protected Web pages.
Connection	This header indicates whether the client can handle persistent HTTP connections. Persistent connections permit the client or

	other browser to retrieve multiple files with a single request. A value of Keep-Alive means that persistent connections should be used
Content-Length	This header is applicable only to POST requests and gives the size of the POST data in bytes.
Cookie	This header returns cookies to servers that previously sent them to the browser.
Host	This header specifies the host and port as given in the original URL.
If-Modified-Since	This header indicates that the client wants the page only if it has been changed after the specified date. The server sends a code, 304 which means Not Modified header if no newer result is available.
If-Unmodified-Since	This header is the reverse of If-Modified-Since; it specifies that the operation should succeed only if the document is older than the specified date.
Referer	This header indicates the URL of the referring Web page. For example, if you are at Web page 1 and click on a link to Web page 2, the URL of Web page 1 is included in the Referer header when the browser requests Web page 2.
User-Agent	This header identifies the browser or other client making the request and can be used to return different content to different types of browsers.

### Methods to read HTTP Header:

There are following methods which can be used to read HTTP header in your servlet program. These methods are available with *HttpServletRequest* object.

#### S.N. Method & Description

- 1 `Cookie[] getCookies()`  
Returns an array containing all of the Cookie objects the client sent with this request.
- 2 `Enumeration getAttributeNames()`  
Returns an Enumeration containing the names of the attributes available to this request.
- 3 `Enumeration getHeaderNames()`  
Returns an enumeration of all the header names this request contains.
- 4 `Enumeration getParameterNames()`  
Returns an Enumeration of String objects containing the names of the parameters contained in this request.
- 5 `HttpSession getSession()`  
Returns the current session associated with this request, or if the request does not have a session, creates one.
- 6 `HttpSession getSession(boolean create)`  
Returns the current HttpSession associated with this request or, if if there is no current session and create is true, returns a new session.

- Locale getLocale()  
7 Returns the preferred Locale that the client will accept content in, based on the Accept-Language header.
- Object getAttribute(String name)  
8 Returns the value of the named attribute as an Object, or null if no attribute of the given name exists.
- ServletInputStream getInputStream()  
9 Retrieves the body of the request as binary data using a ServletInputStream.
- String getAuthType()  
10 Returns the name of the authentication scheme used to protect the servlet, for example, "BASIC" or "SSL," or null if the JSP was not protected.
- String getCharacterEncoding()  
11 Returns the name of the character encoding used in the body of this request.
- String getContentType()  
12 Returns the MIME type of the body of the request, or null if the type is not known.
- String getContextPath()  
13 Returns the portion of the request URI that indicates the context of the request.
- String getHeader(String name)  
14 Returns the value of the specified request header as a String.
- String getMethod()  
15 Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT.
- String getParameter(String name)  
16 Returns the value of a request parameter as a String, or null if the parameter does not exist.
- String getPathInfo()  
17 Returns any extra path information associated with the URL the client sent when it made this request.
- String getProtocol()  
18 Returns the name and version of the protocol the request.
- String getQueryString()  
19 Returns the query string that is contained in the request URL after the path.
- String getRemoteAddr()  
20 Returns the Internet Protocol (IP) address of the client that sent the request.
- String getRemoteHost()  
21 Returns the fully qualified name of the client that sent the request.
- String getRemoteUser()  
22 Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated.
- String getRequestURI()  
23 Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.
- String getRequestedSessionId()  
24 Returns the session ID specified by the client.
- String getServletPath()  
25 Returns the part of this request's URL that calls the JSP.

- String[] getParameterValues(String name)
- 26 Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist.
- boolean isSecure()
- 27 Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS.
- int getContentLength()
- 28 Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known.
- int getIntHeader(String name)
- 29 Returns the value of the specified request header as an int.
- int getServerPort()
- 30 Returns the port number on which this request was received.

### HTTP Header Request Example:

Following is the example which uses **getHeaderNames()** method of `HttpServletRequest` to read the HTTP header information.

This method returns an Enumeration that contains the header information associated with the current HTTP request. Once we have an Enumeration, we can loop down the Enumeration in the standard manner, 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 DisplayHeader 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 = "HTTP Header Request Example";

        String docType = "<!doctype html public \"-//w3c//dtd html 4.0 \" +
            \"transitional//en\">\n";
        out.println(docType +
            "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +
            "<body bgcolor=\"#f0f0f0\">\n" +
            "<h1 align=\"center\">" + title + "</h1>\n" +
```



```

        "<table width=\"100%\" border=\"1\" align=\"center\">\n" +
        "<tr bgcolor=\"#949494\">\n" +
        "<th>Header Name</th><th>Header Value(s)</th>\n"+
        "</tr>\n");
Enumeration headerNames = request.getHeaderNames();
while(headerNames.hasMoreElements()) {
    String paramName = (String)headerNames.nextElement();
    out.print("<tr><td>" + paramName + "</td>\n");
    String paramValue = request.getHeader(paramName);
    out.println("<td> " + paramValue + "</td></tr>\n");
}

out.println("</table>\n</body></html>");
}
// Method to handle POST method request.
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
}

```

### Server HTTP Response

When a Web server responds to a HTTP request to the browser, the response typically consists of a status line, some response headers, a blank line, and the document. A typical response looks like this:

```

HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
(Blank Line)
<!doctype ...>
<html>
  <head>...</head>
  <body>
    ...
  </body>
</html>

```

The status line consists of the HTTP version (HTTP/1.1 in the example), a status code (200 in the example), and a very short message corresponding to the status code (OK in the example). Following is a summary of the most useful HTTP 1.1 response headers which go back to the browser from web server side and you would use them very frequently in web programming:

Header

Allow

Cache-Control

#### Description

This header specifies the request methods (GET, POST, etc.) that the server supports.

This header specifies the circumstances in which the response document can safely be cached. It can have values

	public, private or no-cache etc. Public means document is cacheable, Private means document is for a single user and can only be stored in private (nonshared) caches and no-cache means document should never be cached.
Connection	This header instructs the browser whether to use persistent in HTTP connections or not. A value of close instructs the browser not to use persistent HTTP connections and keep-alive means using persistent connections.
Content-Disposition	This header lets you request that the browser ask the user to save the response to disk in a file of the given name.
Content-Encoding	This header specifies the way in which the page was encoded during transmission.
Content-Language	This header signifies the language in which the document is written. For example en, en-us, ru, etc.
Content-Length	This header indicates the number of bytes in the response. This information is needed only if the browser is using a persistent (keep-alive) HTTP connection.
Content-Type	This header gives the MIME (Multipurpose Internet Mail Extension) type of the response document.
Expires	This header specifies the time at which the content should be considered out-of-date and thus no longer be cached.
Last-Modified	This header indicates when the document was last changed. The client can then cache the document and supply a date by an If-Modified-Since request header in later requests.
Location	This header should be included with all responses that have a status code in the 300s. This notifies the browser of the document address. The browser automatically reconnects to this location and retrieves the new document.
Refresh	This header specifies how soon the browser should ask for an updated page. You can specify time in number of seconds after which a page would be refreshed.
Retry-After	This header can be used in conjunction with a 503 (Service Unavailable) response to tell the client how soon it can repeat its request.
Set-Cookie	This header specifies a cookie associated with the page.

### Methods to Set HTTP Response Header:

There are following methods which can be used to set HTTP response header in your servlet program. These methods are available with *HttpServletResponse* object.

#### S.N. Method & Description

- 1 String encodeRedirectURL(String url)  
Encodes the specified URL for use in the sendRedirect method or, if encoding is not needed, returns the URL unchanged.
- 2 String encodeURL(String url)  
Encodes the specified URL by including the session ID in it, or, if encoding is

- not needed, returns the URL unchanged.
- boolean containsHeader(String name)
- 3 Returns a boolean indicating whether the named response header has already been set.
- boolean isCommitted()
- 4 Returns a boolean indicating if the response has been committed.
- void addCookie(Cookie cookie)
- 5 Adds the specified cookie to the response.
- void addDateHeader(String name, long date)
- 6 Adds a response header with the given name and date-value.
- void addHeader(String name, String value)
- 7 Adds a response header with the given name and value.
- void addIntHeader(String name, int value)
- 8 Adds a response header with the given name and integer value.
- void flushBuffer()
- 9 Forces any content in the buffer to be written to the client.
- void reset()
- 10 Clears any data that exists in the buffer as well as the status code and headers.
- void resetBuffer()
- 11 Clears the content of the underlying buffer in the response without clearing headers or status code.
- void sendError(int sc)
- 12 Sends an error response to the client using the specified status code and clearing the buffer.
- void sendError(int sc, String msg)
- 13 Sends an error response to the client using the specified status.
- void sendRedirect(String location)
- 14 Sends a temporary redirect response to the client using the specified redirect location URL.
- void setBufferSize(int size)
- 15 Sets the preferred buffer size for the body of the response.
- void setCharacterEncoding(String charset)
- 16 Sets the character encoding (MIME charset) of the response being sent to the client, for example, to UTF-8.
- void setContentLength(int len)
- 17 Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header.
- void setContentType(String type)
- 18 Sets the content type of the response being sent to the client, if the response has not been committed yet.
- void setDateHeader(String name, long date)
- 19 Sets a response header with the given name and date-value.
- void setHeader(String name, String value)
- 20 Sets a response header with the given name and value.
- void setIntHeader(String name, int value)
- 21 Sets a response header with the given name and integer value.

- 22    void setLocale(Locale loc)  
      Sets the locale of the response, if the response has not been committed yet.
- 23    void setStatus(int sc)  
      Sets the status code for this response.

### HTTP Header Response Example:

You already have seen `setContentType()` method working in previous examples and following example would also use same method, additionally we would use **`setIntHeader()`** method to set Refresh header.

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.util.\*;

// Extend HttpServlet class

public class Refresh extends HttpServlet {

    // Method to handle GET method request.

    public void doGet(HttpServletRequest request, HttpServletResponse response)

        throws ServletException, IOException {

        // Set refresh, autoload time as 5 seconds

        response.setIntHeader("Refresh", 5);

        // Set response content type

        response.setContentType("text/html");

        // Get current time

        Calendar calendar = new GregorianCalendar();

        String am\_pm;

        int hour = calendar.get(Calendar.HOUR);

        int minute = calendar.get(Calendar.MINUTE);

        int second = calendar.get(Calendar.SECOND);

        if(calendar.get(Calendar.AM\_PM) == 0)

            am\_pm = "AM";

        else

            am\_pm = "PM";

        String CT = hour+":"+ minute +":"+ second + " " + am\_pm;

        PrintWriter out = response.getWriter();

        String title = "Auto Refresh Header Setting";

        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" +
        "<p>Current Time is: " + CT + "</p>\n");
    }

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

```

Now calling the above servlet would display current system time after every 5 seconds.

## 2.6 Using Cookies in Servlets

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. Java Servlets transparently supports HTTP cookies.

There are three steps involved in identifying returning users:

- ☐ Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- ☐ Browser stores this information on local machine for future use.
- ☐ When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

### The Anatomy of a Cookie:

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A servlet that sets a cookie might send headers that look something like this:

```

HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;
path=/; domain=tutorialspoint.com
Connection: close
Content-Type: text/html

```

As you can see, the Set-Cookie header contains a name value pair, a GMT date, a path and a domain. The name and value will be URL encoded. The expires field is an instruction to the browser to "forget" the cookie after the given time and date.

If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server. The browser's headers might look something like this:

```

GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126
Accept: image/gif, */*
Accept-Encoding: gzip

```

Accept-Language: en  
Accept-Charset: iso-8859-1,\*,utf-8  
Cookie: name=xyz

A servlet will then have access to the cookie through the request method `request.getCookies()` which returns an array of *Cookie* objects.

### **Servlet Cookies Methods:**

Following is the list of useful methods which you can use while manipulating cookies in servlet.

S.N.	Method & Description
1	<code>public void setDomain(String pattern)</code> This method sets the domain to which cookie applies, for example tutorialspoint.com.
2	<code>public String getDomain()</code> This method gets the domain to which cookie applies, for example tutorialspoint.com.
3	<code>public void setMaxAge(int expiry)</code> This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
4	<code>public int getMaxAge()</code> This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown.
5	<code>public String getName()</code> This method returns the name of the cookie. The name cannot be changed after creation.
6	<code>public void setValue(String newValue)</code> This method sets the value associated with the cookie.
7	<code>public String getValue()</code> This method gets the value associated with the cookie.
8	<code>public void setPath(String uri)</code> This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.
9	<code>public String getPath()</code> This method gets the path to which this cookie applies.
10	<code>public void setSecure(boolean flag)</code> This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections.
11	<code>public void setComment(String purpose)</code> This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.

12

```
public String getComment()
```

This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.

### Setting Cookies with Servlet:

Setting cookies with servlet involves three steps:

**Creating a Cookie object:** You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

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

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

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

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

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

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

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

Example:

```
import java.io.*;
```

```
import javax.servlet.*;
```

```
import javax.servlet.http.*;
```

```
// Extend HttpServlet class
```

```
public class HelloForm extends HttpServlet {
```

```
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
```

```
        throws ServletException, IOException
```

```
{
```

```
    // Create cookies for first and last names.
```

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

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

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

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

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

```
    // Add both the cookies in the response header.
```

```
    response.addCookie( firstName );
```

```
    response.addCookie( lastName );
```

```
    // Set response content type
```

```
    response.setContentType("text/html");
```

```
    PrintWriter out = response.getWriter();
```

```
    String title = "Setting Cookies Example";
```



```

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

```

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

```

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

```

Keep above HTML content in a file Hello.htm and put it in <Tomcat-installation-directory>/webapps/ROOT directory. When you would access <http://localhost:8080/Hello.htm>,

## 2.7 Session Tracking

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

Still there are following three ways to maintain session between web client and web server:

A web server can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie. This may not be an effective way because many time browser does not support a cookie, so I would not recommend 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 POST data. Each time when web browser sends request back, then session\_id value can be used to keep the track of different web browsers.

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

### URL Rewriting:

You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session. For example, with `http://tutorialspoint.com/file.htm;sessionid=12345`, the session identifier is attached as `sessionid=12345` which can be accessed at the web server to identify the client. URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies but here drawback is that you would have generate every URL dynamically to assign a session ID though page is simple static HTML page.

### The HttpSession Object:

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

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

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

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

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

#### S.N. Method & Description

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

- the client chooses not to join the session.
- ```
public void removeAttribute(String name)
```
- 9 This method removes the object bound with the specified name from this session.
- ```
public void setAttribute(String name, Object value)
```
- 10 This method binds an object to this session, using the name specified.
- ```
public void setMaxInactiveInterval(int interval)
```
- 11 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.

```
// Import required java libraries
```

```
import java.io.*;
```

```
import javax.servlet.*;
```

```
import javax.servlet.http.*;
```

```
import java.util.*;
```

```
// Extend HttpServlet class
```

```
public class SessionTrack extends HttpServlet {
```

```
    public void doGet(HttpServletRequestRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException
    {
```

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

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

```
        // Get session creation time.
```

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

```
        // Get last access time of this web page.
```

```
        Date lastAccessTime =
```

```
            new Date(session.getLastAccessedTime());
```

```
        String title = "Welcome Back to my website";
```

```
        Integer visitCount = new Integer(0);
```

```
        String visitCountKey = new String("visitCount");
```

```
        String userIDKey = new String("userID");
```

```
        String userID = new String("ABCD");
```

```
        // Check if this is new comer on your web page.
```

```
        if (session.isNew()){
```

```
            title = "Welcome to my website";
```

```
            session.setAttribute(userIDKey, userID);
```

```
        } else {
```

```
            visitCount = (Integer)session.getAttribute(visitCountKey);
```

```
            visitCount = visitCount + 1;
```

```

        userID = (String)session.getAttribute(userIDKey);
    }
    session.setAttribute(visitCountKey, visitCount);
    // Set response content type
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
    "<!doctype html public \"-//w3c//dtd html 4.0 \" +
    \"transitional//en\">\n";
    out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<h2 align=\"center\">Session Infomation</h2>\n" +
        "<table border=\"1\" align=\"center\">\n" +
        "<tr bgcolor=\"#949494\">\n" +
        " <th>Session info</th><th>value</th></tr>\n" +
        "<tr>\n" +
        " <td>id</td>\n" +
        " <td>" + session.getId() + "</td></tr>\n" +
        "<tr>\n" +
        " <td>Creation Time</td>\n" +
        " <td>" + createTime +
        " </td></tr>\n" +
        "<tr>\n" +
        " <td>Time of Last Access</td>\n" +
        " <td>" + lastAccessTime +
        " </td></tr>\n" +
        "<tr>\n" +
        " <td>User ID</td>\n" +
        " <td>" + userID +
        " </td></tr>\n" +
        "<tr>\n" +
        " <td>Number of visits</td>\n" +
        " <td>" + visitCount + "</td></tr>\n" +
        "</table>\n" +
        "</body></html>");
    }
}

```

Compile above servlet SessionTrack and create appropriate entry in web.xml file.  
Now running is <http://localhost:8080/SessionTrack>.

## 2.8 Servlet Database

To start with basic concept, let us create a simple table and create few records in that table as follows:

### Create Table

To create the Employees table in TEST database, use the following steps:

### Step 1:

Open a Command Prompt and change to the installation directory as follows:

```
C:\>
```

```
C:\>cd Program Files\MySQL\bin
```

```
C:\Program Files\MySQL\bin>
```

### Step 2:

Login to database as follows

```
C:\Program Files\MySQL\bin>mysql -u root -p
```

```
Enter password: *****
```

```
mysql>
```

### Step 3:

Create the table Employee in TEST database as follows:

```
mysql> use TEST;
```

```
mysql> create table Employees
```

```
(
    id int not null,
    age int not null,
    first varchar (255),
    last varchar (255)
);
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
mysql>
```

Create Data Records

Finally you create few records in Employee table as follows:

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
```

```
Query OK, 1 row affected (0.05 sec)
```

```
mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
```

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

```
mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
```

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

```
mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
```

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

```
mysql>
```

### Accessing a Database:

Here is an example which shows how to access TEST database using Servlet.

```
// Loading required libraries
```

```
import java.io.*;
```

```
import java.util.*;
```

```
import javax.servlet.*;
```

```
import javax.servlet.http.*;
```

```
import java.sql.*;
```

```
public class DatabaseAccess extends HttpServlet{
```

```
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
```

```

        throws ServletException, IOException
    {
        // JDBC driver name and database URL
        static final String JDBC_DRIVER="com.mysql.jdbc.Driver";
        static final String DB_URL="jdbc:mysql://localhost/TEST";

        // Database credentials
        static final String USER = "root";
        static final String PASS = "password";

        // Set response content type
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Database Result";
        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");
        try{
            // Register JDBC driver
            Class.forName("com.mysql.jdbc.Driver");

            // Open a connection
            Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);

            // Execute SQL query
            Statement stmt = conn.createStatement();
            String sql;
            sql = "SELECT id, first, last, age FROM Employees";
            ResultSet rs = stmt.executeQuery(sql);

            // Extract data from result set
            while(rs.next()){
                //Retrieve by column name
                int id  = rs.getInt("id");
                int age = rs.getInt("age");
                String first = rs.getString("first");
                String last = rs.getString("last");

                //Display values
                out.println("ID: " + id + "<br>");
                out.println("Age: " + age + "<br>");
                out.println("First: " + first + "<br>");
                out.println("Last: " + last + "<br>");
            }
        }
    }

```

```

        out.println("</body></html>");

        // Clean-up environment
        rs.close();
        stmt.close();
        conn.close();
    } catch(SQLException se){
        //Handle errors for JDBC
        se.printStackTrace();
    } catch(Exception e){
        //Handle errors for Class.forName
        e.printStackTrace();
    } finally{
        //finally block used to close resources
        try{
            if(stmt!=null)
                stmt.close();
        } catch(SQLException se2){
        } // nothing we can do
        try{
            if(conn!=null)
                conn.close();
        } catch(SQLException se){
            se.printStackTrace();
        } //end finally try
    } //end try
}
}

```

Now let us compile above servlet and create following entries in web.xml

```

....
<servlet>
    <servlet-name>DatabaseAccess</servlet-name>
    <servlet-class>DatabaseAccess</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>DatabaseAccess</servlet-name>
    <url-pattern>/DatabaseAccess</url-pattern>
</servlet-mapping>
....

```

Now call this servlet using URL <http://localhost:8080/DatabaseAccess> which would display following response:

Database Result

```

ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal

```

JSP: Overview of JSP Technology, Need of JSP, Advantages of JSP, Life Cycle of JSP Page, JSP Processing, JSP Application Design with MVC, Setting Up the JSP Environment, JSP Directives, JSP Action, JSP Implicit Objects, JSP Form Processing, JSP Session and Cookies Handling, JSP Session Tracking JSP Database Access, JSP Standard Tag Libraries, JSP Custom Tag, JSP Expression Language, JSP Exception Handling

### JAVA SERVER PAGES

#### 3.1 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.

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

A Java Server Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.

Using JSP, you can collect input from users through Webpage forms, present records from a database or another source, and create Webpages dynamically.

JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages, and sharing information between requests, pages etc.

#### 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.



### 3.2 Need and Importance of JSP

Java Server Pages often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But JSP offers several advantages in comparison with the CGI. Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself instead of having separate CGI files.

JSP are always compiled before they are processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested. Java Server Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including JDBC, JNDI, EJB, JAXP, etc.

JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

Finally, JSP is an integral part of Java EE, a complete platform for enterprise class applications. This means that JSP can play a part in the simplest applications to the most complex and demanding.

#### Advantages of JSP

Following table lists out the other advantages of using JSP over other technologies –

##### vs. Active Server Pages (ASP)

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

##### vs. Pure Servlets

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

##### vs. Server-Side Includes (SSI)

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

##### vs. JavaScript

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

##### vs. Static HTML

Regular HTML, of course, cannot contain dynamic information.

### 3.3 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 Figure 3.1, 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.

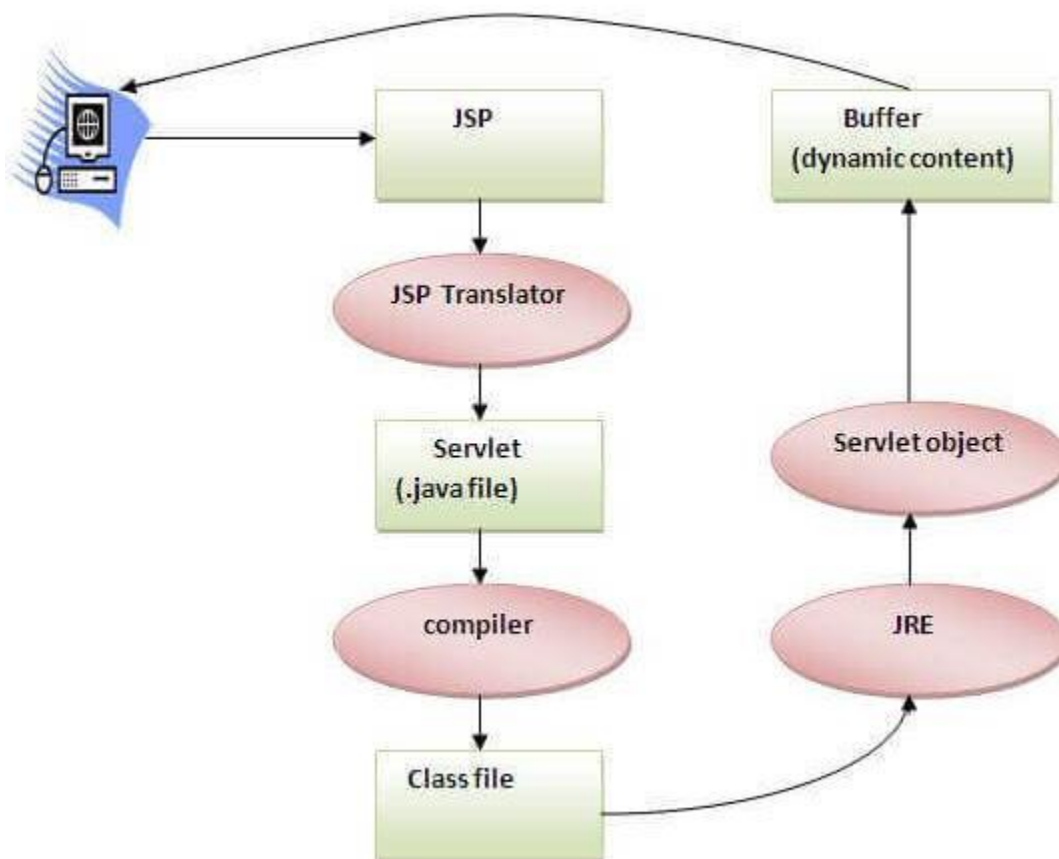
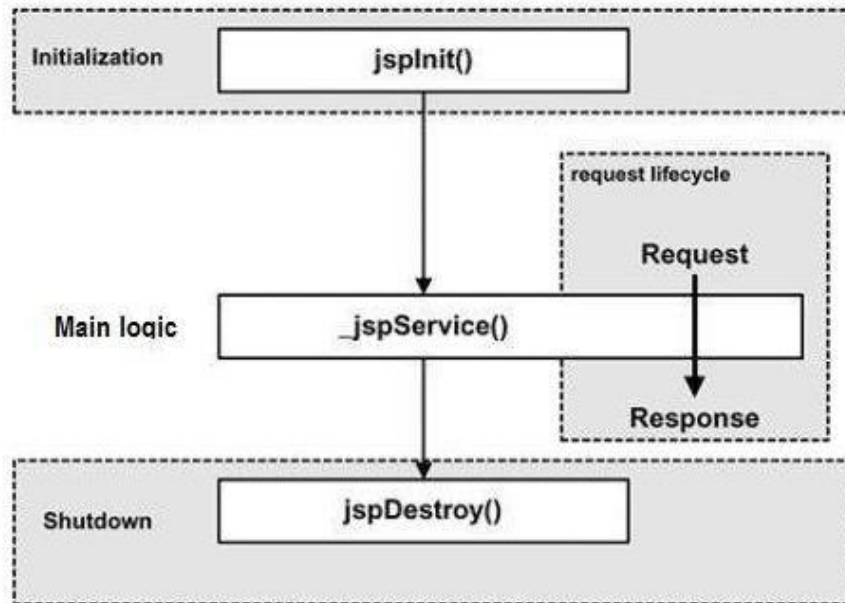
A JSP life cycle is defined as the process from its creation till the destruction. This is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet.

### **Paths Followed By JSP**

The following are the paths followed by a JSP –

- 1.Compilation
- 2.Initialization
- 3.Execution
- 4.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 –



**Figure 3.1: JSP Architecture**

### 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.  
}
```

## 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>
```

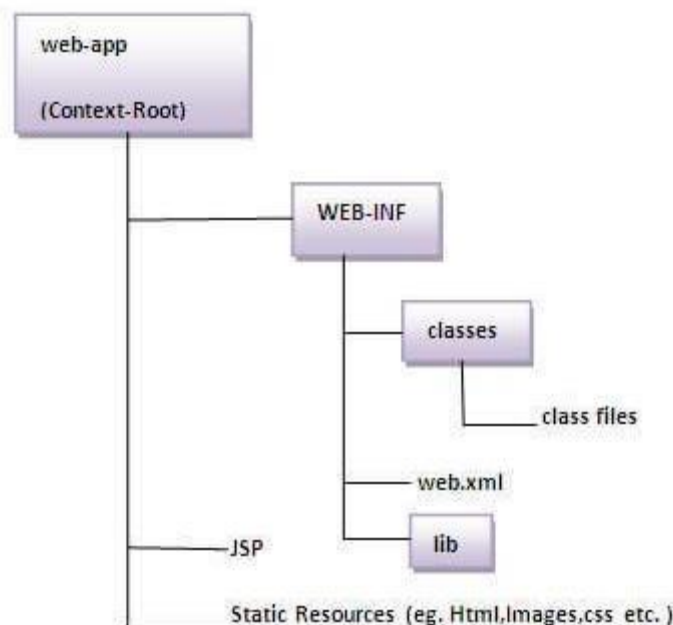
Follow the following steps to execute this JSP page:

- 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`

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.



**Figure 3.2:** Director Structure of JSP

## 3.4 MVC Architecture in JSP

What is MVC?

MVC is an architecture that separates business logic, presentation and data. In MVC,

M stands for Model

V stands for View

C stands for controller.

MVC is a systematic way to use the application where the flow starts from the view layer, where the request is raised and processed in controller layer and sent to model layer to insert data and get back the success or failure message.

#### **Model Layer:**

This is the data layer which consists of the business logic of the system. It consists of all the data of the application

It also represents the state of the application.

It consists of classes which have the connection to the database.

The controller connects with model and fetches the data and sends to the view layer.

The model connects with the database as well and stores the data into a database which is connected to it.

#### **View Layer:**

This is a presentation layer.

It consists of HTML, JSP, etc. into it.

It normally presents the UI of the application.

It is used to display the data which is fetched from the controller which in turn fetching data from model layer classes.

This view layer shows the data on UI of the application.

#### **Controller Layer:**

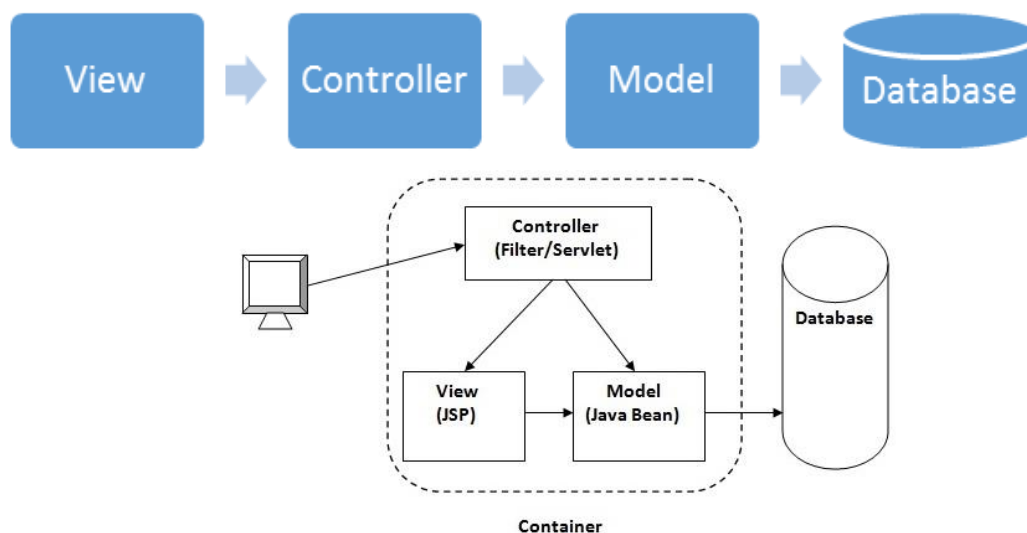
It acts as an interface between View and Model.

It intercepts all the requests which are coming from the view layer.

It receives the requests from the view layer and processes the requests and does the necessary validation for the request.

This request is further sent to model layer for data processing, and once the request is processed, it sends back to the controller with required information and displayed accordingly by the view.

The diagram is represented below:



**Figure 3.3:** MVC Architecture of JSP

#### **The advantages of MVC are:**

Easy to maintain

Easy to extend

Easy to test

Navigation control is centralized

## Example of MVC architecture

In this example, we are going to show how to use MVC architecture in JSP. We are taking the example of a form with two variables "email" and "password" which is our view layer.

Once the user enters email, and password and clicks on submit then the action is passed in mvc\_servlet where email and password are passed.

This mvc\_servlet is controller layer. Here in mvc\_servlet the request is sent to the bean object which act as model layer.

The email and password values are set into the bean and stored for further purpose. From the bean, the value is fetched and shown in the view layer.

Mvc\_example.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">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>MVC Guru Example</title>
</head>
<body>
<form action="Mvc_servlet" method="POST">
Email: <input type="text" name="email">
<br />
Password: <input type="text" name="password" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

### Explanation of the code:

#### View Layer:

**Code Line 10-15:** Here we are taking a form which has two fields as parameter "email" and "password" and this request need to be forwarded to a controller Mvc\_servlet.java, which is passed in action. The method through which it is passed is POST method.

Mvc\_servlet.java

```
package demotest;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class Mvc_servlet
 */
public class Mvc_servlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
```

```

    * @see HttpServlet#HttpServlet()
    */
    public Mvc_servlet() {
        super();
        // TODO Auto-generated constructor stub
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        // TODO Auto-generated method stub
        String email=request.getParameter("email");
        String password=request.getParameter("password");

        TestBean testobj = new TestBean();
        testobj.setEmail(email);
        testobj.setPassword(password);
        request.setAttribute("gurubean",testobj);
        RequestDispatcher rd=request.getRequestDispatcher("mvc_success.jsp");
        rd.forward(request, response);
    }
}

```

### Controller layer

**Code Line 14:** mvc\_servlet is extending HttpServlet.

**Code Line 26:** As the method used is POST hence request comes into a doPost method of the servlet which process the requests and saves into the bean object as testobj.

**Code Line 34:** Using request object we are setting the attribute as gurubean which is assigned the value of testobj.

**Code Line 35:** Here we are using request dispatcher object to pass the success message to mvc\_success.jsp

TestBean.java

package demotest;

import java.io.Serializable;

```

public class TestBean implements Serializable{
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    private String email="null";
    private String password="null";
}

```

**Explanation of the code:**

**Model Layer:**



**Code Line 7-17:** It contains the getters and setters of email and password which are members of Test Bean class

**Code Line 19-20:** It defines the members email and password of string type in the bean class.

Mvc\_success.jsp

```
<% @ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"% >
    <% @page import="demotest.TestBean"% >
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Guru Success</title>
</head>
<body>
<%
TestBean testguru=(TestBean)request.getAttribute("gurubean");
out.print("Welcome, "+testguru.getEmail());
%>
</body>
</html>
```

#### Explanation of the code:

**Code Line 12:** we are getting the attribute using request object which has been set in the doPost method of the servlet.

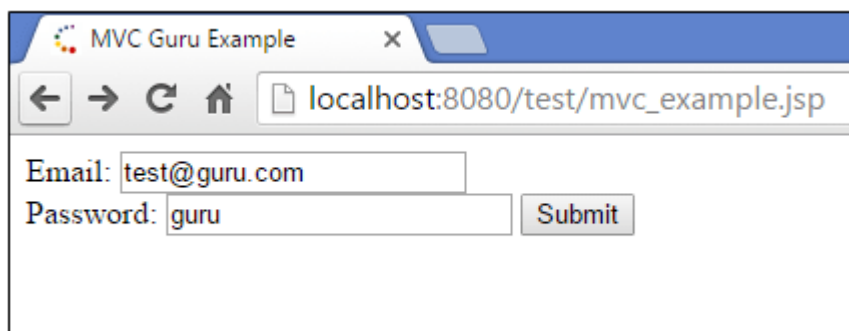
**Code Line 13:** We are printing the welcome message and email id of which have been saved in the bean object

#### Output:

When you execute the above code, you get the following output:

When you click on mvc\_example.jsp you get the form with email and password with the submit button.

Once you enter email and password to the form and then click on submit

A screenshot of a web browser window titled "MVC Guru Example". The address bar shows "localhost:8080/test/mvc\_example.jsp". The form contains two input fields: "Email:" with the value "test@guru.com" and "Password:" with the value "guru". A "Submit" button is located to the right of the password field.

After clicking on submit the output is shown as below

**Output:**

When you enter email and password in screen and click on submit then, the details are saved in TestBean and from the TestBean they are fetched on next screen to get the success message.

**3.5 JSP Environments and Directives**

A development environment is where you would develop your JSP programs, test them and finally run them.

This tutorial will guide you to setup your JSP development environment which involves the following steps –

**Setting up Java Development Kit**

This step involves downloading an implementation of the Java Software Development Kit (SDK) and setting up the PATH environment variable appropriately.

Once you download your Java implementation, follow the given instructions to install and configure the setup. Finally set the PATH and JAVA\_HOME environment variables to refer to the directory that contains java and javac, typically java\_install\_dir/bin and java\_install\_dir respectively.

If you are running Windows and install the SDK in C:\jdk1.5.0\_20, you need to add the following line in your C:\autoexec.bat file.

```
set PATH = C:\jdk1.5.0_20\bin;%PATH%
```

```
set JAVA_HOME = C:\jdk1.5.0_20
```

Alternatively, on Windows NT/2000/XP, you can also right-click on My Computer, select Properties, then Advanced, followed by Environment Variables. Then, you would update the PATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if the SDK is installed in /usr/local/jdk1.5.0\_20 and you use the C shell, you will put the following into your .cshrc file.

```
setenv PATH /usr/local/jdk1.5.0_20/bin:$PATH
```

```
setenv JAVA_HOME /usr/local/jdk1.5.0_20
```

Alternatively, if you use an Integrated Development Environment (IDE) like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, compile and run a simple program to confirm that the IDE knows where you installed Java.

**Setting up Web Server: Tomcat**

A number of Web Servers that support Java Server Pages and Servlets development are available in the market. Some web servers can be downloaded for free and Tomcat is one of them.

Apache Tomcat is an open source software implementation of the Java Server Pages and Servlet technologies and can act as a standalone server for testing JSP and Servlets, and can be integrated with the Apache Web Server. Here are the steps to set up Tomcat on your machine –

Download the latest version of Tomcat from <https://tomcat.apache.org/>.

Once you downloaded the installation, unpack the binary distribution into a convenient location. For example, in C:\apache-tomcat-5.5.29 on windows, or /usr/local/apache-tomcat-5.5.29 on Linux/Unix and create CATALINA\_HOME environment variable pointing to these locations.

Tomcat can be started by executing the following commands on the Windows machine – %CATALINA\_HOME%\bin\startup.bat

or

C:\apache-tomcat-5.5.29\bin\startup.bat

Tomcat can be started by executing the following commands on the Unix (Solaris, Linux, etc.) machine –

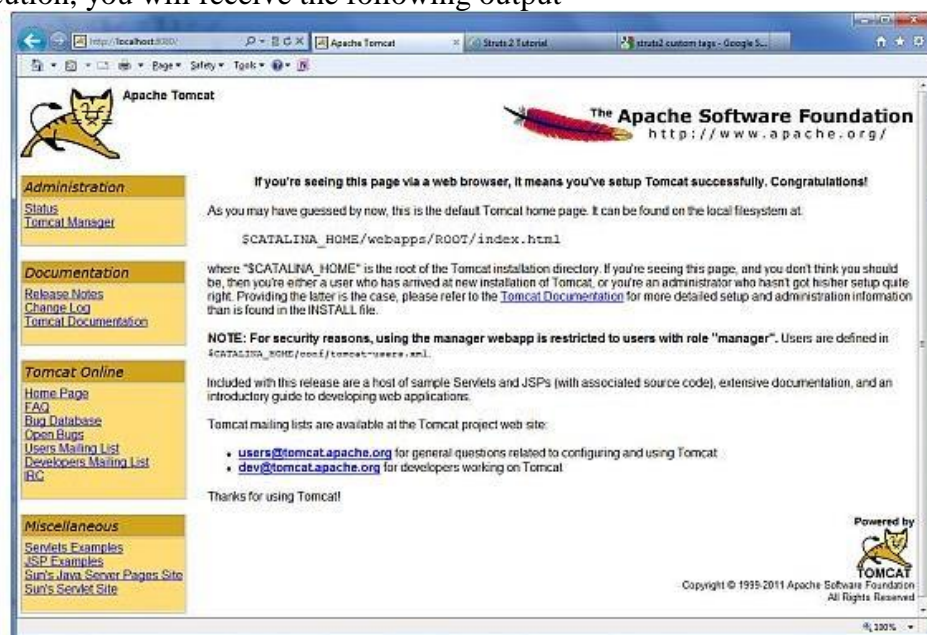
\$CATALINA\_HOME/bin/startup.sh

or

/usr/local/apache-tomcat-5.5.29/bin/startup.sh

After a successful startup, the default web-applications included with Tomcat will be available by visiting <http://localhost:8080/>.

Upon execution, you will receive the following output –



Further information about configuring and running Tomcat can be found in the documentation included here, as well as on the Tomcat web site – <https://tomcat.apache.org/>.

Tomcat can be stopped by executing the following commands on the Windows machine – %CATALINA\_HOME%\bin\shutdown

or

C:\apache-tomcat-5.5.29\bin\shutdown

Tomcat can be stopped by executing the following commands on Unix (Solaris, Linux, etc.) machine –

\$CATALINA\_HOME/bin/shutdown.sh

or

/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh

## Setting up CLASSPATH

Since servlets are not part of the Java Platform, Standard Edition, you must identify the servlet classes to the compiler.

If you are running Windows, you need to put the following lines in your C:\autoexec.bat file.  
set CATALINA = C:\apache-tomcat-5.5.29

```
set CLASSPATH = %CATALINA%\common\lib\jsp-api.jar;%CLASSPATH%
```

Alternatively, on Windows NT/2000/XP, you can also right-click on My Computer, select Properties, then Advanced, then Environment Variables. Then, you would update the CLASSPATH value and press the OK button.

On Unix (Solaris, Linux, etc.), if you are using the C shell, you would put the following lines into your .cshrc file.

```
setenv CATALINA = /usr/local/apache-tomcat-5.5.29
```

```
setenv CLASSPATH $CATALINA/common/lib/jsp-api.jar:$CLASSPATH
```

NOTE – Assuming that your development directory is C:\JSPDev (Windows) or /usr/JSPDev (Unix), then you would need to add these directories as well in CLASSPATH.

## JSP Directives

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

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

Directives can have a number of attributes which you can list down as key-value pairs and separated by commas.

The blanks between the @ symbol and the directive name, and between the last attribute and the closing %>, are optional.

There are three types of directive tag –

S.No.	Directive & Description
1	<pre>&lt;%@ page ... %&gt;</pre> Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
2	<pre>&lt;%@ include ... %&gt;</pre> Includes a file during the translation phase.
3	<pre>&lt;%@ taglib ... %&gt;</pre> 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" %>
```

You can write the XML equivalent of the above syntax as follows –

```
<jsp:directive.page attribute = "value" />
```

Attributes

Following table lists out the attributes associated with the page directive –

S.No.	Attribute & Purpose
1	Buffer Specifies a buffering model for the output stream.
2	autoFlush Controls the behavior of the servlet output buffer.

3	contentType Defines the character encoding scheme.
4	errorPage Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
5	isErrorPage Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.
6	Extends Specifies a superclass that the generated servlet must extend.
7	Import Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
8	Info Defines a string that can be accessed with the servlet's getServletInfo() method.
9	isThreadSafe Defines the threading model for the generated servlet.
10	Language Defines the programming language used in the JSP page.
11	Session Specifies whether or not the JSP page participates in HTTP sessions
12	isELIgnored Specifies whether or not the EL expression within the JSP page will be ignored.
13	isScriptingEnabled Determines if the scripting elements are allowed for use.

### 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" />
```

The filename in the include directive is actually a relative URL. If you just specify a filename with no associated path, the JSP compiler assumes that the file is in the same directory as your JSP.

You can write the XML equivalent of the above syntax as follows –

```
<jsp:directive.include file = "relative url" />
```

#### Example

A good example of the include directive is including a common header and footer with multiple pages of content.

Let us define following three files (a) header.jsp, (b) footer.jsp, and (c) main.jsp as follows – Following is the content of header.jsp –

```
<%!
    int pageCount = 0;
    void addCount() {
        pageCount++;
    }
%>
```

```

<% addCount(); %>

<html>
  <head>
    <title>The include Directive Example</title>
  </head>

  <body>
    <center>
      <h2>The include Directive Example</h2>
      <p>This site has been visited <%= pageCount %> times.</p>
    </center>
    <br/><br/>

```

Following is the content of footer.jsp –

```

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

```

Finally here is the content of main.jsp –

```

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

```

Let us now keep all these files in the root directory and try to access main.jsp. You will receive the following output –

The include Directive Example

This site has been visited 1 times.

Thanks for visiting my page.

Refresh main.jsp and you will find that the page hit counter keeps increasing. You can design your webpages based on your creative instincts; it is recommended you keep the dynamic parts of your website in separate files and then include them in the main file. This makes it easy when you need to change a part of your webpage.

### 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" >

```

Here, the uri attribute value resolves to a location the container understands and the prefix attribute informs a container what bits of markup are custom actions.

You can write the XML equivalent of the above syntax as follows –

```
<jsp:directive.taglib uri = "uri" prefix = "prefixOfTag" />
```

Refresh main.jsp and you will find that the page hit counter keeps increasing. You can design your webpages based on your creative instincts; it is recommended you keep the dynamic parts of your website in separate files and then include them in the main file. This makes it easy when you need to change a part of your webpage.

Example

For example, suppose the custlib tag library contains a tag called hello. If you wanted to use the hello tag with a prefix of mytag, your tag would be <mytag:hello> and it will be used in your JSP file as follows –

```
<%@ taglib uri = "http://www.example.com/custlib" prefix = "mytag" %>
```

```
<html>
  <body>
    <mytag:hello/>
  </body>
</html>
```

We can call another piece of code using <mytag:hello>.

### 3.6 JSP Actions and Implicit Objects

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.

There is only one syntax for the Action element, as it conforms to the XML standard – Action elements are basically predefined functions. The following table lists out the available JSP actions –

S. No.	Syntax & Purpose
1	jsp:include Includes a file at the time the page is requested.
2	jsp:useBean Finds or instantiates a JavaBean.
3	jsp:setProperty Sets the property of a JavaBean.
4	jsp:getProperty Inserts the property of a JavaBean into the output.
5	jsp:forward Forwards the requester to a new page.
6	jsp:plugin Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin.
7	jsp:element Defines XML elements dynamically.
8	jsp:attribute Defines dynamically-defined XML element's attribute.
9	jsp:body Defines dynamically-defined XML element's body.
10	jsp:text Used to write template text in JSP pages and documents.

**Example**

Let us define the following two files (a)date.jsp and (b) main.jsp as follows  
– Following is the content of the date.jsp file –

## Common Attributes

There are two attributes that are common to all Action elements: the id attribute and the scope attribute.

### Id attribute

The id attribute uniquely identifies the Action element, and allows the action to be referenced inside the JSP page. If the Action creates an instance of an object, the id value can be used to reference it through the implicit object PageContext.

### Scope attribute

This attribute identifies the lifecycle of the Action element. The id attribute and the scope attribute are directly related, as the scope attribute determines the lifespan of the object associated with the id. The scope attribute has four possible values: (a) page, (b)request, (c)session, and (d) application.

### The <jsp:include> Action

This action lets you insert files into the page being generated. The syntax looks like this

– `<jsp:include page = "relative URL" flush = "true" />`

Unlike the include directive, which inserts the file at the time the JSP page is translated into a servlet, this action inserts the file at the time the page is requested.

Following table lists out the attributes associated with the include action –

S.No.	Attribute & Description
1	page The relative URL of the page to be included.
2	flush The boolean attribute determines whether the included resource has its buffer flushed before it is included.

## Example

Let us define the following two files (a)date.jsp and (b) main.jsp as follows

– Following is the content of the date.jsp file –

```
<p>Today's date: <%= (new java.util.Date()).toLocaleString()%></p>
```

Following is the content of the main.jsp file –

```
<html>
  <head>
    <title>The include Action Example</title>
  </head>

  <body>
    <center>
      <h2>The include action Example</h2>
      <jsp:include page = "date.jsp" flush = "true" />
    </center>
  </body>
</html>
```

Let us now keep all these files in the root directory and try to access main.jsp. You will receive the following output –

```
<html>
```

The include action Example



### The <jsp:useBean> Action

The useBean action is quite versatile. It first searches for an existing object utilizing the id and scope variables. If an object is not found, it then tries to create the specified object.

The simplest way to load a bean is as follows –

```
<jsp:useBean id = "name" class = "package.class" />
```

Once a bean class is loaded,

you can use jsp:setProperty and jsp:getProperty actions to modify and retrieve the bean properties.

Following table lists out the attributes associated with the useBean action –

.No.	Attribute & Description
1	class Designates the full package name of the bean.
2	type Specifies the type of the variable that will refer to the object.
3	beanName Gives the name of the bean as specified by the instantiate () method of the java.beans.Beans class.

Let us now discuss the jsp:setProperty and the jsp:getProperty actions before giving a valid example related to these actions.

### The <jsp:setProperty> Action

The setProperty action sets the properties of a Bean. The Bean must have been previously defined before this action. There are two basic ways to use the setProperty action –

You can use jsp:setProperty after, but outside of a jsp:useBean element, as given below –

```
<jsp:useBean id = "myName" ... />
```

...

```
<jsp:setProperty name = "myName" property = "someProperty" .../>
```

In this case, the jsp:setProperty is executed regardless of whether a new bean was instantiated or an existing bean was found.

A second context in which jsp:setProperty can appear is inside the body of a jsp:useBean element, as given below –

```
<jsp:useBean id = "myName" ... >
```

...

```
<jsp:setProperty name = "myName" property = "someProperty" .../>
```

```
</jsp:useBean>
```

Here, the jsp:setProperty is executed only if a new object was instantiated, not if an existing one was found.

Following table lists out the attributes associated with the setProperty action –

S.No.	Attribute & Description
1	Name Designates the bean the property of which will be set. The Bean must have been previously defined.
2	Property Indicates the property you want to set. A value of "*" means that all request parameters whose names match bean property names will be passed to the appropriate setter methods.
3	value The value that is to be assigned to the given property. The the parameter's value is null, or the parameter does not exist, the setProperty action is ignored.
4	Param The param attribute is the name of the request parameter whose value the property is to receive. You can't use both value and param, but it is permissible to use neither.

### The <jsp:getProperty> Action

The getProperty action is used to retrieve the value of a given property and converts it to a string, and finally inserts it into the output.

The getProperty action has only two attributes, both of which are required. The syntax of the getProperty action is as follows –

```
<jsp:useBean id = "myName" ... />
```

...

```
<jsp:getProperty name = "myName" property = "someProperty" .../>
```

Following table lists out the required attributes associated with the getProperty action –

S.No.	Attribute & Description
1	name The name of the Bean that has a property to be retrieved. The Bean must have been previously defined.
2	property The property attribute is the name of the Bean property to be retrieved.

### Example

Let us define a test bean that will further be used in our example –

```
/* File: TestBean.java */
```

```
package action;
```

```
/* File: TestBean.java */
```

```
package action;
```

```
public class TestBean {
    private String message = "No message specified";

    public String getMessage() {
        return(message);
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

Compile the above code to the generated TestBean.class file and make sure that you copied the TestBean.class in C:\apache-tomcat-7.0.2\webapps\WEB-INF\classes\action folder and the CLASSPATH variable should also be set to this folder –

Now use the following code in main.jsp file. This loads the bean and sets/gets a simple String parameter –

```
<html>

<head>
  <title>Using JavaBeans in JSP</title>
</head>

<body>
  <center>
    <h2>Using JavaBeans in JSP</h2>
    <jsp:useBean id = "test" class = "action.TestBean" />
    <jsp:setProperty name = "test" property = "message"
      value = "Hello JSP..." />

    <p>Got message....</p>
    <jsp:getProperty name = "test" property = "message" />
  </center>
</body>
</html>
```

Let us now try to access main.jsp, it would display the following result –

Using JavaBeans in JSP

Got message....

Hello JSP...

The <jsp:forward> Action

The forward action terminates the action of the current page and forwards the request to another resource such as a static page, another JSP page, or a Java Servlet.

Following is the syntax of the forward action –

```
<jsp:forward page = "Relative URL" />
```

Following table lists out the required attributes associated with the forward action –

S.No.	Attribute & Description
1	page Should consist of a relative URL of another resource such as a static page, another JSP page, or a Java Servlet.

Example

Let us reuse the following two files (a) date.jsp and (b) main.jsp as follows –

Following is the content of the date.jsp file –

```
<p>Today's date: <%= (new java.util.Date()).toLocaleString()%></p>
```

Following is the content of the main.jsp file –

```
<html>
<head>
  <title>The include Action Example</title>
</head>

<body>
```

```

<center>
  <h2>The include action Example</h2>
  <jsp:forward page = "date.jsp" />
</center>
</body>
</html>

```

Let us now keep all these files in the root directory and try to access main.jsp. This would display result something like as below.

Here it discarded the content from the main page and displayed the content from forwarded page only.

Today's date: 12-Sep-2010 14:54:22

### The <jsp:plugin> Action

The plugin action is used to insert Java components into a JSP page. It determines the type of browser and inserts the <object> or <embed> tags as needed.

If the needed plugin is not present, it downloads the plugin and then executes the Java component. The Java component can be either an Applet or a JavaBean.

The plugin action has several attributes that correspond to common HTML tags used to format Java components. The <param> element can also be used to send parameters to the Applet or Bean.

Following is the typical syntax of using the plugin action –

```

<jsp:plugin type = "applet" codebase = "dirname" code = "MyApplet.class"
  width = "60" height = "80">
  <jsp:param name = "fontcolor" value = "red" />
  <jsp:param name = "background" value = "black" />

  <jsp:fallback>
    Unable to initialize Java Plugin
  </jsp:fallback>

```

```

</jsp:plugin>

```

You can try this action using some applet if you are interested. A new element, the <fallback> element, can be used to specify an error string to be sent to the user in case the component fails.

The <jsp:element> Action

The <jsp:attribute> Action

The <jsp:body> Action

The <jsp:element>, <jsp:attribute> and <jsp:body> actions are used to define XML elements dynamically. The word dynamically is important, because it means that the XML elements can be generated at request time rather than statically at compile time.

Following is a simple example to define XML elements dynamically –

```

<% @page language = "java" contentType = "text/html"%>
<html xmlns = "http://www.w3c.org/1999/xhtml"
  xmlns:jsp = "http://java.sun.com/JSP/Page">
  <head><title>Generate XML Element</title></head>
  <body>
    <jsp:element name = "xmlElement">
      <jsp:attribute name = "xmlElementAttr">
        Value for the attribute
      </jsp:attribute>
    </jsp:element>
  </body>
</html>

```

```

    </jsp:attribute>

    <jsp:body>
        Body for XML element
    </jsp:body>

    </jsp:element>
</body>
</html>
This would produce the following HTML code at run time –
<html xmlns = "http://www.w3c.org/1999/xhtml" xmlns:jsp =
"http://java.sun.com/JSP/Page">
    <head><title>Generate XML Element</title></head>

    <body>
        <xmlElement xmlElementAttr = "Value for the attribute">
            Body for XML element
        </xmlElement>
    </body>
</html>

```

### The <jsp:text> Action

The <jsp:text> action can be used to write the template text in JSP pages and documents. Following is the simple syntax for this action –

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

The body of the template cannot contain other elements; it can only contain text and EL expressions (Note – EL expressions are explained in a subsequent chapter). Note that in XML files, you cannot use expressions such as `${whatever > 0}`, because the greater than signs are illegal. Instead, use the `gt` form, such as `${whatever gt 0}` or an alternative is to embed the value in a CDATA section.

```
<jsp:text><![CDATA[<br>]]></jsp:text>
```

If you need to include a DOCTYPE declaration, for instance for XHTML, you must also use the <jsp:text> element as follows –

```
<jsp:text><![CDATA[<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml11-strict.dtd">]]></jsp:text>
```

```

<head><title>jsp:text action</title></head>

<body>
    <books><book><jsp:text>
        Welcome to JSP Programming
    </jsp:text></book></books>
</body>
</html>

```

### JSP Implicit Object

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 table lists out the nine Implicit Objects that JSP supports –

S.No.	Object & Description
1	Request This is the <code>HttpServletRequest</code> object associated with the request.
2	Response This is the <code>HttpServletResponse</code> object associated with the response to the client.
3	Out This is the <code>PrintWriter</code> object used to send output to the client.
4	Session This is the <code>HttpSession</code> object associated with the request.
5	Application This is the <code>ServletContext</code> object associated with the application context.
6	Config This is the <code>ServletConfig</code> object associated with the page.
7	pageContext This encapsulates use of server-specific features like higher performance <code>JspWriters</code> .
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 <code>Exception</code> 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 the 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 table lists out the important methods that we will use to write boolean, char, int, double, object, String, etc.

S.No.	Method & Description
1	<code>out.print(dataType dt)</code> Print a data type value
2	<code>out.println(dataType dt)</code> Print a data type value then terminate the line with new line character.
3	<code>out.flush()</code> Flush the stream.

### The session Object

The session object is an instance of `javax.servlet.http.HttpSession` and behaves exactly the same way that session objects behave under Java Servlets.

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

### The application Object

The application object is direct wrapper around the `ServletContext` object for the generated Servlet and in reality an instance of a `javax.servlet.ServletContext` object.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the `jspDestroy()` method.

By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it.

### The config Object

The config object is an instantiation of `javax.servlet.ServletConfig` and is a direct wrapper around the `ServletConfig` object for the generated servlet.

This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc.

The following config method is the only one you might ever use, and its usage is trivial – `config.getServletName();`

This returns the servlet name, which is the string contained in the `<servlet-name>` element defined in the `WEB-INF\web.xml` file.

### The pageContext Object

The `pageContext` object is an instance of a `javax.servlet.jsp.PageContext` object.

The `pageContext` object is used to represent the entire JSP page.

This object is intended as a means to access information about the page while avoiding most of the implementation details.

This object stores references to the request and response objects for each request.

The application, config, session, and out objects are derived by accessing attributes of this object.

The `pageContext` object also contains information about the directives issued to the JSP page, including the buffering information, the `errorPageURL`, and page scope.

The `PageContext` class defines several fields, including `PAGE_SCOPE`, `REQUEST_SCOPE`, `SESSION_SCOPE`, and `APPLICATION_SCOPE`, which identify the four scopes. It also supports more than 40 methods, about half of which are inherited from the `javax.servlet.jsp.JspContext` class.

One of the important methods is `removeAttribute`. This method accepts either one or two arguments. For example, `pageContext.removeAttribute("attrName")` removes the attribute from all scopes, while the following code only removes it from the page scope – `pageContext.removeAttribute("attrName", PAGE_SCOPE);`

### The page Object

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page.

The page object is really a direct synonym for the this object. The exception Object

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

### 3.7 JSP Session and Cookies Handling

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 –



.No.	Method & Description
1	<code>public Object getAttribute(String name)</code> This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	<code>public Enumeration getAttributeNames()</code> This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3	<code>public long getCreationTime()</code> This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4	<code>public String getId()</code> This method returns a string containing the unique identifier assigned to this session.
5	<code>public long getLastAccessedTime()</code> 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	<code>public int getMaxInactiveInterval()</code> This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
7	<code>public void invalidate()</code> This method invalidates this session and unbinds any objects bound to it.
8	<code>public boolean isNew()</code> 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	<code>public void removeAttribute(String name)</code> This method removes the object bound with the specified name from this session.
10	<code>public void setAttribute(String name, Object value)</code> This method binds an object to this session, using the name specified.
11	<code>public void setMaxInactiveInterval(int interval)</code> 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>
    </table>

```

```

<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>

```

Now put the above code in main.jsp and try to access <http://localhost:8080/main.jsp>. Once you run the URL, you will receive the following result –

Welcome to my website  
Session Information

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

Now try to run the same JSP for the second time, you will receive the following result.

Welcome Back to my website  
Session Information

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

### Deleting Session Data

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

Remove a particular attribute – You can call the *public void removeAttribute(String name)* method to delete the value associated with the a particular key.

Delete the whole session – You can call the *public void invalidate()* method to discard an entire session.

Setting Session timeout – You can call the *public void setMaxInactiveInterval(int interval)* method to set the timeout for a session individually.

Log the user out – The servers that support servlets 2.4, you can call *logout* to log the client out of the Web server and invalidate all sessions belonging to all the users.

web.xml Configuration – If you are using Tomcat, apart from the above mentioned methods, you can configure the session time out in web.xml file as follows.

```
<session-config>
  <session-timeout>15</session-timeout>
</session-config>
```

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

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

### JSP Cookies Handling

Cookies are text files stored on the client computer and they are kept for various information tracking purposes. JSP transparently supports HTTP cookies using underlying servlet technology.

There are three steps involved in identifying and returning users –

Server script sends a set of cookies to the browser. For example, name, age, or identification number, etc.

Browser stores this information on the local machine for future use.

When the next time the browser sends any request to the web server then it sends those cookies information to the server and server uses that information to identify the user or may be for some other purpose as well.

This chapter will teach you how to set or reset cookies, how to access them and how to delete them using JSP programs.

#### The Anatomy of a Cookie

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A JSP that sets a cookie might send headers that look something like this –

HTTP/1.1 200 OK

Date: Fri, 04 Feb 2000 21:03:38 GMT

Server: Apache/1.3.9 (UNIX) PHP/4.0b3

Set-Cookie: name = xyz; expires = Friday, 04-Feb-07 22:03:38 GMT;  
path = /; domain = tutorialspoint.com

Connection: close

Content-Type: text/html

As you can see, the Set-Cookie header contains a name value pair, a GMT date, a path and a domain. The name and value will be URL encoded. The expires field is an instruction to the browser to "forget" the cookie after the given time and date.

If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server. The browser's headers might look something like this –

GET / HTTP/1.0

Connection: Keep-Alive

User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)

Host: zink.demon.co.uk:1126

Accept: image/gif, \*/\*  
Accept-Encoding: gzip  
Accept-Language: en  
Accept-Charset: iso-8859-1,\*,utf-8  
Cookie: name = xyz

A JSP script will then have access to the cookies through the request method *request.getCookies()* which returns an array of *Cookie* objects.

#### Servlet Cookies Methods

Following table lists out the useful methods associated with the Cookie object which you can use while manipulating cookies in JSP –

S.No.	Method & Description
1	<code>public void setDomain(String pattern)</code> This method sets the domain to which the cookie applies; for example, <code>tutorialspoint.com</code> .
2	<code>public String getDomain()</code> This method gets the domain to which the cookie applies; for example, <code>tutorialspoint.com</code> .
3	<code>public void setMaxAge(int expiry)</code> This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
4	<code>public int getMaxAge()</code> This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until the browser shutdown.
5	<code>public String getName()</code> This method returns the name of the cookie. The name cannot be changed after the creation.
6	<code>public void setValue(String newValue)</code> This method sets the value associated with the cookie.
7	<code>public String getValue()</code> This method gets the value associated with the cookie.
8	<code>public void setPath(String uri)</code> This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.
9	<code>public String getPath()</code> This method gets the path to which this cookie applies.
10	<code>public void setSecure(boolean flag)</code> This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e, SSL) connections.

11	<pre>public void setComment(String purpose)</pre> <p>This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.</p>
12	<pre>public String getComment()</pre> <p>This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.</p>

### Setting Cookies with JSP

Setting cookies with JSP involves three steps –

Step 1: Creating a Cookie object

You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

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

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

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

Step 3: Sending the Cookie into the HTTP response headers

You use response.addCookie to add cookies in the HTTP response header as follows

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

Example

set the cookies for the first and the last name.

```
<%
```

```
// 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>

```

Let us put the above code in main.jsp file and use it in the following HTML page

```

<html>
<body>

    <form action = "main.jsp" method = "GET">
        First Name: <input type = "text" name = "first_name">
        <br />
        Last Name: <input type = "text" name = "last_name" />
        <input type = "submit" value = "Submit" />
    </form>

</body>
</html>

```

Keep the above HTML content in a file hello.jsp and put hello.jsp and main.jsp in <Tomcat-installation-directory>/webapps/ROOT directory. When you will access <http://localhost:8080/hello.jsp>, here is the actual output of the above form.



The screenshot shows the rendered HTML form. It has two text input fields, one for 'First Name' and one for 'Last Name'. To the right of the 'Last Name' field is a button labeled 'Submit'.

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

In the next section, we will explain how you can access these cookies back in your web application.

### Reading Cookies with JSP

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

#### Example

Let us now read cookies that were set in the previous example –

```

<html>
<head>
    <title>Reading Cookies</title>
</head>

<body>
    <center>
        <h1>Reading Cookies</h1>
    </center>
    <%
        Cookie cookie = null;
        Cookie[] cookies = null;

```

```

// Get an array of Cookies associated with the this domain
cookies = request.getCookies();

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

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

</html>

```

Let us now put the above code in main.jsp file and try to access it. If you set the first\_name cookie as "John" and the last\_name cookie as "Player" then running *http://localhost:8080/main.jsp* will display the following result –

```

Found Cookies Name and Value
Name : first_name, Value: John
Name : last_name, Value: Player

```

### **Delete Cookies with JSP**

To delete cookies is very simple. If you want to delete a cookie, then you simply need to follow these three steps –

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

Set cookie age as zero using the setMaxAge() method to delete an existing cookie.

Add this cookie back into the response header.

Example

Following example will show you how to delete an existing cookie named "first\_name" and when you run main.jsp JSP next time, it will return null value for first\_name.

```

<html>
<head>
    <title>Reading Cookies</title>
</head>

<body>
    <center>
        <h1>Reading Cookies</h1>
    </center>
    <%
        Cookie cookie = null;
        Cookie[] cookies = null;

        // Get an array of Cookies associated with the this domain
        cookies = request.getCookies();
    %>

```



```

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

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

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

```

</html>

Let us now put the above code in the main.jsp file and try to access it. It will display the following result –

Cookies Name and Value

Deleted cookie : first\_name

Name : first\_name, Value: John

Name : last\_name, Value: Player

Now run *http://localhost:8080/main.jsp* once again and it should display only one cookie as follows

Found Cookies Name and Value

Name : last\_name, Value: Player

You can delete your cookies in the Internet Explorer manually. Start at the Tools menu and select the Internet Options. To delete all cookies, click the Delete Cookies button.

### 3.8 Database Access and JSP Standard Tag Libraries

To start with basic concept, let us create a table and create a few records in that table as follows

#### Create Table

To create the **Employees** table in the EMP database, use the following steps –

##### Step 1

Open a **Command Prompt** and change to the installation directory as follows –

C:\>

C:\>cd Program Files\MySQL\bin

C:\Program Files\MySQL\bin>

##### Step 2

Login to the database as follows –

C:\Program Files\MySQL\bin>mysql -u root -p

Enter password: \*\*\*\*\*

mysql>

### Step 3

Create the **Employee** table in the **TEST** database as follows – –

```
mysql> use TEST;
mysql> create table Employees
(
    id int not null,
    age int not null,
    first varchar (255),
    last varchar (255)
);
Query OK, 0 rows affected (0.08 sec)
mysql>
```

#### Create Data Records

Let us now create a few records in the **Employee** table as follows – –

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)

mysql>
```

### SELECT Operation

Following example shows how we can execute the **SQL SELECT** statement using JTSL in JSP programming –

```
<% @ page import = "java.io.*,java.util.*,java.sql.*"%>
<% @ page import = "javax.servlet.http.*,javax.servlet.*" %>
<% @ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<% @ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
<head>
    <title>SELECT Operation</title>
</head>

<body>
    <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
        url = "jdbc:mysql://localhost/TEST"
        user = "root" password = "pass123"/>

    <sql:query dataSource = "${snapshot}" var = "result">
        SELECT * from Employees;
    </sql:query>
```

```

<table border = "1" width = "100%">
  <tr>
    <th>Emp ID</th>
    <th>First Name</th>
    <th>Last Name</th>
    <th>Age</th>
  </tr>

  <c:forEach var = "row" items = "${result.rows}">
    <tr>
      <td><c:out value = "${row.id}"/></td>
      <td><c:out value = "${row.first}"/></td>
      <td><c:out value = "${row.last}"/></td>
      <td><c:out value = "${row.age}"/></td>
    </tr>
  </c:forEach>
</table>

</body>
</html>

```

### INSERT Operation

Following example shows how we can execute the SQL INSERT statement using JSTL in JSP programming –

```

<% @ page import = "java.io.*,java.util.*,java.sql.*"%>
<% @ page import = "javax.servlet.http.*,javax.servlet.*" %>
<% @ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<% @ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
  <head>
    <title>JINSERT Operation</title>
  </head>

  <body>
    <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
      url = "jdbc:mysql://localhost/TEST"
      user = "root" password = "pass123"/>
    <sql:update dataSource = "${snapshot}" var = "result">
      INSERT INTO Employees VALUES (104, 2, 'Nuha', 'Ali');
    </sql:update>

    <sql:query dataSource = "${snapshot}" var = "result">
      SELECT * from Employees;
    </sql:query>

    <table border = "1" width = "100%">
      <tr>
        <th>Emp ID</th>
        <th>First Name</th>

```

```

        <th>Last Name</th>
        <th>Age</th>
    </tr>

    <c:forEach var = "row" items = "${result.rows}">
        <tr>
            <td><c:out value = "${row.id}" /></td>
            <td><c:out value = "${row.first}" /></td>
            <td><c:out value = "${row.last}" /></td>
            <td><c:out value = "${row.age}" /></td>
        </tr>
    </c:forEach>
</table>

</body>
</html>

```

## DELETE Operation

Following example shows how we can execute the **SQL DELETE** statement using JTSL in JSP programming –

```

<% @ page import = "java.io.*,java.util.*,java.sql.*"%>
<% @ page import = "javax.servlet.http.*,javax.servlet.*" %>
<% @ taglib uri="http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<% @ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
<head>
    <title>DELETE Operation</title>
</head>

<body>
    <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
        url = "jdbc:mysql://localhost/TEST"
        user = "root" password = "pass123"/>

    <c:set var = "empId" value = "103"/>

    <sql:update dataSource = "${snapshot}" var = "count">
        DELETE FROM Employees WHERE Id = ?
        <sql:param value = "${empId}" />
    </sql:update>

    <sql:query dataSource = "${snapshot}" var = "result">
        SELECT * from Employees;
    </sql:query>

    <table border = "1" width = "100%">
        <tr>
            <th>Emp ID</th>
            <th>First Name</th>

```

```

        <th>Last Name</th>
        <th>Age</th>
    </tr>

    <c:forEach var = "row" items = "${result.rows}">
        <tr>
            <td><c:out value = "${row.id}" /></td>
            <td><c:out value = "${row.first}" /></td>
            <td><c:out value = "${row.last}" /></td>
            <td><c:out value = "${row.age}" /></td>
        </tr>
    </c:forEach>
</table>

```

```

</body>
</html>

```

## UPDATE Operation

Following example shows how we can execute the **SQL UPDATE** statement using JTSL in JSP programming –

```

<% @ page import = "java.io.*,java.util.*,java.sql.*"%>
<% @ page import = "javax.servlet.http.*,javax.servlet.*" %>
<% @ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<% @ taglib uri = "http://java.sun.com/jsp/jstl/sql" prefix = "sql"%>

<html>
<head>
    <title>DELETE Operation</title>
</head>

<body>
    <sql:setDataSource var = "snapshot" driver = "com.mysql.jdbc.Driver"
        url = "jdbc:mysql://localhost/TEST"
        user = "root" password = "pass123"/>

    <c:set var = "empId" value = "102"/>

    <sql:update dataSource = "${snapshot}" var = "count">
        UPDATE Employees SET WHERE last = 'Ali'
        <sql:param value = "${empId}" />
    </sql:update>

    <sql:query dataSource = "${snapshot}" var = "result">
        SELECT * from Employees;
    </sql:query>

    <table border = "1" width = "100%">
        <tr>
            <th>Emp ID</th>
            <th>First Name</th>
            <th>Last Name</th>

```

```

        <th>Age</th>
    </tr>

    <c:forEach var = "row" items = "${result.rows}">
        <tr>
            <td><c:out value = "${row.id}"/></td>
            <td><c:out value = "${row.first}"/></td>
            <td><c:out value = "${row.last}"/></td>
            <td><c:out value = "${row.age}"/></td>
        </tr>
    </c:forEach>
</table>

</body>
</html>

```

## JSP Standard Tag Libraries

The JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates the 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 the existing custom tags with the JSTL tags.

### Install JSTL Library

To begin working with JSP tags you need to first install the JSTL library. If you are using the Apache Tomcat container, then follow these two steps –

**Step 1** – Download the binary distribution from [Apache Standard Taglib](#) and unpack the compressed file.

**Step 2** – To use the Standard Taglib from its **Jakarta Taglibs distribution**, simply copy the JAR files in the distribution's 'lib' directory to your application's **webapps\ROOT\WEB-INF\lib** directory.

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

### Classification of The JSTL Tags

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

#### Core Tags

#### Formatting tags

#### SQL tags

#### XML tags

#### JSTL Functions

#### Core Tags

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

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

Following table lists out the core JSTL Tags –

S.No.	Tag & Description
1	<u>&lt;c:out&gt;</u> Like <%= ... >, but for expressions.

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

### Formatting Tags

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

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

Following table lists out the Formatting JSTL Tags –

S.No.	Tag & Description
-------	-------------------

1	<u>&lt;fmt:formatNumber&gt;</u> To render numerical value with specific precision or format.
2	<u>&lt;fmt:parseNumber&gt;</u> Parses the string representation of a number, currency, or percentage.
3	<u>&lt;fmt:formatDate&gt;</u> Formats a date and/or time using the supplied styles and pattern.
4	<u>&lt;fmt:parseDate&gt;</u> Parses the string representation of a date and/or time
5	<u>&lt;fmt:bundle&gt;</u> Loads a resource bundle to be used by its tag body.
6	<u>&lt;fmt:setLocale&gt;</u> Stores the given locale in the locale configuration variable.
7	<u>&lt;fmt:setBundle&gt;</u> Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable.
8	<u>&lt;fmt:timeZone&gt;</u> Specifies the time zone for any time formatting or parsing actions nested in its body.
9	<u>&lt;fmt:setTimeZone&gt;</u> Stores the given time zone in the time zone configuration variable
10	<u>&lt;fmt:message&gt;</u> Displays an internationalized message.
11	<u>&lt;fmt:requestEncoding&gt;</u> Sets the request character encoding

### 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 table lists out the SQL JSTL Tags –

S.No.	Tag & Description
1	<u>&lt;sql:setDataSource&gt;</u> Creates a simple DataSource suitable only for prototyping
2	<u>&lt;sql:query&gt;</u> Executes the SQL query defined in its body or through the sql attribute.
3	<u>&lt;sql:update&gt;</u>



	Executes the SQL update defined in its body or through the sql attribute.
4	<sql:param> Sets a parameter in an SQL statement to the specified value.
5	<sql:dateParam> Sets a parameter in an SQL statement to the specified java.util.Date value.
6	<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 the XML documents. Following is the syntax to include the JSTL XML library in your JSP.

The JSTL XML tag library has custom tags for interacting with the XML data. This includes parsing the XML, transforming the XML data, and the flow control based on the XPath expressions.

<% @ taglib prefix = "x"

uri = "http://java.sun.com/jsp/jstl/xml" %>

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

**XercesImpl.jar** – Download it from <https://www.apache.org/dist/xerces/j/>

**xalan.jar** – Download it from <https://xml.apache.org/xalan-j/index.html>

Following is the list of XML JSTL Tags –

S.No.	Tag & Description
1	<x:out> Like <%= ... >, but for XPath expressions.
2	<x:parse> Used to parse the XML data specified either via an attribute or in the tag body.
3	<x:set > Sets a variable to the value of an XPath expression.
4	<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.
5	<x:forEach> To loop over nodes in an XML document.
6	<x:choose> Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise> tags.
7	<x:when > Subtag of <choose> that includes its body if its expression evalutes to 'true'.

8	<x:otherwise > Subtag of <choose> that follows the <when> tags and runs only if all of the prior conditions evaluates to 'false'.
9	<x:transform > Applies an XSL transformation on a XML document
10	<x:param > Used along with the transform tag to set a parameter in the XSLT stylesheet

### 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 table lists out the various JSTL Functions –

S.No.	Function & Description
1	fn:contains() Tests if an input string contains the specified substring.
2	fn:containsIgnoreCase() Tests if an input string contains the specified substring in a case insensitive way.
3	fn:endsWith() Tests if an input string ends with the specified suffix.
4	fn:escapeXml() Escapes characters that can be interpreted as XML markup.
5	fn:indexOf() Returns the index withing a string of the first occurrence of a specified substring.
6	fn:join() Joins all elements of an array into a string.
7	fn:length() Returns the number of items in a collection, or the number of characters in a string.
8	fn:replace() Returns a string resulting from replacing in an input string all occurrences with a given string.
9	fn:split() Splits a string into an array of substrings.
10	fn:startsWith() Tests if an input string starts with the specified prefix.
11	fn:substring() Returns a subset of a string.
12	fn:substringAfter() Returns a subset of a string following a specific substring.

13	fn:substringBefore() Returns a subset of a string before a specific substring.
14	fn:toLowerCase() Converts all of the characters of a string to lower case.
15	fn:toUpperCase() Converts all of the characters of a string to upper case.
16	fn:trim() Removes white spaces from both ends of a string.

### 3.9 JSP Custom Tag, JSP Expression Language, and JSP Exception Handling

A custom tag is a user-defined JSP language element. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The Web container then invokes those operations when the JSP page's servlet is executed.

JSP tag extensions lets you create new tags that you can insert directly into a JavaServer Page. The JSP 2.0 specification introduced the Simple Tag Handlers for writing these custom tags. To write a custom tag, you can simply extend **SimpleTagSupport** class and override the **doTag()** method, where you can place your code to generate content for the tag.

#### Create "Hello" Tag

Consider you want to define a custom tag named `<ex:Hello>` and you want to use it in the following fashion without a body –

```
<ex:Hello />
```

To create a custom JSP tag, you must first create a Java class that acts as a tag handler. Let us now create the **HelloTag** class as follows –

```
package com.tutorialspoint;
```

```
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;
```

```
public class HelloTag extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        JspWriter out = getJspContext().getOut();
        out.println("Hello Custom Tag!");
    }
}
```

The above code has simple coding where the **doTag()** method takes the current **JspContext** object using the **getJspContext()** method and uses it to send **"Hello Custom Tag!"** to the current **JspWriter** object

Let us compile the above class and copy it in a directory available in the environment variable CLASSPATH. Finally, create the following tag library file: **<Tomcat-Installation-Directory>webapps\ROOT\WEB-INF\custom.tld**.

```
<taglib>
<tlib-version>1.0</tlib-version>
```

```
<jsp-version>2.0</jsp-version>
<short-name>Example TLD</short-name>
```

```
<tag>
  <name>Hello</name>
  <tag-class>com.tutorialspoint.HelloTag</tag-class>
  <body-content>empty</body-content>
</tag>
</taglib>
```

Let us now use the above defined custom tag **Hello** in our JSP program as follows –

```
<% @ taglib prefix = "ex" uri = "WEB-INF/custom.tld"%>
```

```
<html>
  <head>
    <title>A sample custom tag</title>
  </head>

  <body>
    <ex:Hello/>
  </body>
</html>
```

Call the above JSP and this should produce the following result –  
Hello Custom Tag!

### Accessing the Tag Body

You can include a message in the body of the tag as you have seen with standard tags. Consider you want to define a custom tag named **<ex:Hello>** and you want to use it in the following fashion with a body –

```
<ex:Hello>
```

This is message body

```
</ex:Hello>
```

Let us make the following changes in the above tag code to process the body of the tag –  
package com.tutorialspoint;

```
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {
  StringWriter sw = new StringWriter();
  public void doTag()

  throws JspException, IOException {
    getJspBody().invoke(sw);
    getJspContext().getOut().println(sw.toString());
  }
}
```

Here, the output resulting from the invocation is first captured into a **StringWriter** before being written to the JspWriter associated with the tag. We need to change TLD file as follows –

```

<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD with Body</short-name>

```

```

  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
</taglib>

```

Let us now call the above tag with proper body as follows –

```

<% @ taglib prefix = "ex" uri = "WEB-INF/custom.tld"%>

```

```

<html>
  <head>
    <title>A sample custom tag</title>
  </head>

```

```

  <body>
    <ex:Hello>
      This is message body
    </ex:Hello>
  </body>
</html>

```

You will receive the following result –

This is message body

### Custom Tag Attributes

You can use various attributes along with your custom tags. To accept an attribute value, a custom tag class needs to implement the **setter** methods, identical to the JavaBean setter methods as shown below –

```

package com.tutorialspoint;

```

```

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

```

```

public class HelloTag extends SimpleTagSupport {
  private String message;

```

```

  public void setMessage(String msg) {
    this.message = msg;
  }

```

```

  StringWriter sw = new StringWriter();
  public void doTag()

```

```

    throws JspException, IOException {
    if (message != null) {
      /* Use message from attribute */
      JspWriter out = getJspContext().getOut();

```

```

        out.println( message );
    } else {
        /* use message from the body */
        getJspBody().invoke(sw);
        getJspContext().getOut().println(sw.toString());
    }
}
}

```

The attribute's name is "**message**", so the setter method is **setMessage()**. Let us now add this attribute in the TLD file using the **<attribute>** element as follows –

```

<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD with Body</short-name>

  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>scriptless</body-content>

    <attribute>
      <name>message</name>
    </attribute>

  </tag>
</taglib>

```

Let us follow JSP with message attribute as follows –  
 <% @ taglib prefix = "ex" uri = "WEB-INF/custom.tld"%>

```

<html>
  <head>
    <title>A sample custom tag</title>
  </head>

  <body>
    <ex:Hello message = "This is custom tag" />
  </body>
</html>

```

This will produce following result

This is custom tag

Consider including the following properties for an attribute –

S.No.	Property & Purpose
1	Name The name element defines the name of an attribute. Each attribute name must be unique for a particular tag.
2	Required This specifies if this attribute is required or is an optional one. It would be false for optional.

3	Rtexprvalue Declares if a runtime expression value for a tag attribute is valid
4	Type Defines the Java class-type of this attribute. By default it is assumed as String
5	Description Informational description can be provided.
6	Fragment Declares if this attribute value should be treated as a JspFragment.

Following is the example to specify properties related to an attribute –

.....

```
<attribute>
  <name>attribute_name</name>
  <required>>false</required>
  <type>java.util.Date</type>
  <fragment>>false</fragment>
</attribute>
```

.....

If you are using two attributes, then you can modify your TLD as follows –

....

```
<attribute>
  <name>attribute_name1</name>
  <required>>false</required>
  <type>java.util.Boolean</type>
  <fragment>>false</fragment>
</attribute>
```

```
<attribute>
  <name>attribute_name2</name>
  <required>>true</required>
  <type>java.util.Date</type>
</attribute>
```

.....

## JSP Expression Language and JSP Exception Handling

access to the header values makes it possible to easily access application data stored in JavaBeans components. JSP EL allows you to create expressions both (a) arithmetic and (b) logical. Within a JSP EL expression, you can use integers, floating point numbers, strings, the built-in constants true and false for boolean values, and null.

### Simple Syntax

Typically, when you specify an attribute value in a JSP tag, you simply use a string. For example – `<jsp:setProperty name = "box" property = "perimeter" value = "100"/>`

JSP EL allows you to specify an expression for any of these attribute values. A simple syntax for JSP EL is as follows –

`${expr}`

Here expr specifies the expression itself. The most common operators in JSP EL are . and []. These two operators allow you to access various attributes of Java Beans and built-in JSP objects.

For example, the above syntax `<jsp:setProperty>` tag can be written with an expression like –

`<jsp:setProperty name = "box" property = "perimeter"  
value = "${2*box.width+2*box.height}"/>`

When the JSP compiler sees the `${ }` form in an attribute, it generates code to evaluate the expression and substitutes the value of expression.

You can also use the JSP EL expressions within template text for a tag. For example, the `<jsp:text>` tag simply inserts its content within the body of a JSP. The following `<jsp:text>` declaration inserts `<h1>Hello JSP!</h1>` into the JSP output

```
<jsp:text>  
  <h1>Hello JSP!</h1>  
</jsp:text>
```

You can now include a JSP EL expression in the body of a `<jsp:text>` tag (or any other tag) with the same `${ }` syntax you use for attributes. For example –

```
<jsp:text>  
  Box Perimeter is: ${2*box.width + 2*box.height}  
</jsp:text>
```

EL expressions can use parentheses to group subexpressions. For example, `${(1 + 2) * 3}` equals 9, but `${1 + (2 * 3)}` equals 7.

To deactivate the evaluation of EL expressions, we specify the `isELIgnored` attribute of the page directive as below –

```
<%@ page isELIgnored = "true|false" %>
```

The valid values of this attribute are true and false. If it is true, EL expressions are ignored when they appear in static text or tag attributes. If it is false, EL expressions are evaluated by the container.

### Basic Operators in EL

JSP Expression Language (EL) supports most of the arithmetic and logical operators supported by Java. Following table lists out the most frequently used operators –

S.No.	Operator & Description
1	. Access a bean property or Map entry
2	[] Access an array or List element
3	( ) Group a subexpression to change the evaluation order



4	+ Addition
5	- Subtraction or negation of a value
6	* Multiplication
7	/ or div Division
8	% or mod Modulo (remainder)
9	== or eq Test for equality
10	!= or ne Test for inequality
11	< or lt Test for less than
12	> or gt Test for greater than
13	<= or le Test for less than or equal
14	>= or ge Test for greater than or equal
15	&& or and Test for logical AND
16	or or Test for logical OR
17	! or not Unary Boolean complement
18	Empty Test for empty variable values

### Functions in JSP EL

JSP EL allows you to use functions in expressions as well. These functions must be defined in the custom tag libraries. A function usage has the following syntax

```
${ns:func(param1, param2, ...)}
```

Where ns is the namespace of the function, func is the name of the function and param1 is the first parameter value. For example, the function fn:length, which is part of the JSTL library. This function can be used as follows to get the length of a string.

```
${fn:length("Get my length")}
```

o use a function from any tag library (standard or custom), you must install that library on your server and must include the library in your JSP using the <taglib> directive as explained in the JSTL chapter.

### JSP EL Implicit Objects

The JSP expression language supports the following implicit objects –

S.No	Implicit object & Description
1	pageScope Scoped variables from page scope
2	requestScope Scoped variables from request scope

3	sessionScope	Scoped variables from session scope
4	applicationScope	Scoped variables from application scope
5	Param	Request parameters as strings
6	paramValues	Request parameters as collections of strings
7	Header	HTTP request headers as strings
8	headerValues	HTTP request headers as collections of strings
9	initParam	Context-initialization parameters
10	Cookie	Cookie values
11	pageContext	The JSP PageContext object for the current page

You can use these objects in an expression as if they were variables. The examples that follow will help you understand the concepts –

### **The pageContext Object**

The pageContext object gives you access to the pageContext JSP object. Through the pageContext object, you can access the request object. For example, to access the incoming query string for a request, you can use the following expression –  
`${pageContext.request.queryString}`

### **The Scope Objects**

The pageScope, requestScope, sessionScope, and applicationScope variables provide access to variables stored at each scope level.

For example, if you need to explicitly access the box variable in the application scope, you can access it through the applicationScope variable as `applicationScope.box`.

### **The param and paramValues Objects**

The param and paramValues objects give you access to the parameter values normally available through the `request.getParameter` and `request.getParameterValues` methods. For example, to access a parameter named order, use the expression `${param.order}` or `${param["order"]}`.

Following is the example to access a request parameter named username –

```
<% @ page import = "java.io.*,java.util.*" %>
<%String title = "Accessing Request Param";%>
```

```
<html>
<head>
<title><% out.print(title); %></title>
</head>

<body>
<center>
<h1><% out.print(title); %></h1>
</center>

<div align = "center">
<p>${param["username"]}</p>
```

```

    </div>
</body>
</html>

```

The param object returns single string values, whereas the paramValues object returns string arrays.

### header and headerValues Objects

The header and headerValues objects give you access to the header values normally available through the request.getHeader and the request.getHeaders methods.

For example, to access a header named user-agent, use the expression `${header.user-agent}` or `${header["user-agent"]}`.

Following is the example to access a header parameter named user-agent –

```

<% @ page import = "java.io.*,java.util.*" %>
<%String title = "User Agent Example";%>

```

```

<html>
<head>
<title><% out.print(title); %></title>
</head>

<body>
<center>
<h1><% out.print(title); %></h1>
</center>

<div align = "center">
<p>${header["user-agent"]}</p>
</div>
</body>
</html>

```

The output will somewhat be like the following

User Agent Example

Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/4.0;  
SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729;  
Media Center PC 6.0; HPNTDF; .NET4.0C; InfoPath.2)

### JSP Exception Handling

When you are writing a JSP code, you might make coding errors which can occur at any part of the code. There may occur the following type of errors in your JSP code –

#### Checked exceptions

A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

#### Runtime exceptions

A runtime exception is an exception that probably could have been avoided by the programmer. As opposed to the checked exceptions, runtime exceptions are ignored at the time of compilation.

#### Errors

These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything

about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

We will further discuss ways to handle run time exception/error occurring in your JSP code.

#### Using Exception Object

The exception object is an instance of a subclass of Throwable (e.g., java.lang.NullPointerException) and is only available in error pages. Following table lists out the important methods available in the Throwable class.

S.No.	Methods & Description
1	<code>public String getMessage()</code> Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	<code>public Throwable getCause()</code> Returns the cause of the exception as represented by a Throwable object.
3	<code>public String toString()</code> Returns the name of the class concatenated with the result of <code>getMessage()</code> .
4	<code>public void printStackTrace()</code> Prints the result of <code>toString()</code> along with the stack trace to <code>System.err</code> , the error output stream.
5	<code>public StackTraceElement [] getStackTrace()</code> Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	<code>public Throwable fillInStackTrace()</code> Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

JSP gives you an option to specify Error Page for each JSP. Whenever the page throws an exception, the JSP container automatically invokes the error page.

Following is an example to specify an error page for a main.jsp. To set up an error page, use the `<%@ page errorPage = "xxx" %>` directive.

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

```
<html>
<head>
  <title>Error Handling Example</title>
</head>

<body>
  <%
    // Throw an exception to invoke the error page
    int x = 1;

    if (x == 1) {
      throw new RuntimeException("Error condition!!!");
    }
  %>
</body>
</html>
```

```

    }
    %>
</body>
</html>

```

We will now write one Error Handling JSP ShowError.jsp, which is given below. Notice that the error-handling page includes the directive `<%@ page isErrorPage = "true" %>`. This directive causes the JSP compiler to generate the exception instance variable.

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

```

<html>
<head>
  <title>Show Error Page</title>
</head>

<body>
  <h1>Opps...</h1>
  <p>Sorry, an error occurred.</p>
  <p>Here is the exception stack trace: </p>
  <pre><% exception.printStackTrace(response.getWriter()); %></pre>
</body>
</html>

```

Access the main.jsp, you will receive an output somewhat like the following –  
java.lang.RuntimeException: Error condition!!!

.....

Opps...  
Sorry, an error occurred.

Here is the exception stack trace:

Using JSTL Tags for Error Page

You can make use of JSTL tags to write an error page ShowError.jsp. This page has almost same logic as in the above example, with better structure and more information `<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>`

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

```

<html>
<head>
  <title>Show Error Page</title>
</head>

<body>
  <h1>Opps...</h1>
  <table width = "100%" border = "1">
    <tr valign = "top">
      <td width = "40%"><b>Error:</b></td>
      <td>${pageContext.exception}</td>
    </tr>

    <tr valign = "top">

```

```

        <td><b>URI:</b></td>
        <td>${pageContext.errorData.requestURI}</td>
    </tr>

    <tr valign = "top">
        <td><b>Status code:</b></td>
        <td>${pageContext.errorData.statusCode}</td>
    </tr>

    <tr valign = "top">
        <td><b>Stack trace:</b></td>
        <td>
            <c:forEach var = "trace"
                items = "${pageContext.exception.stackTrace}">
                <p>${trace}</p>
            </c:forEach>
        </td>
    </tr>
</table>

</body>
</html>

```

Access the main.jsp, the following will be generated –

Error:	java.lang.RuntimeException: Error condition!!!
URI:	/main.jsp
Status code:	500
Stack trace:	org.apache.jsp.main_jsp._jspService(main_jsp.java:65) org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:68) javax.servlet.http.HttpServlet.service(HttpServlet.java:722) org.apache.jasper.servlet.JspServlet.service(JspServlet.java:265) javax.servlet.http.HttpServlet.service(HttpServlet.java:722)

#### Using Try...Catch Block

If you want to handle errors within the same page and want to take some action instead of firing an error page, you can make use of the try....catch block.

Following is a simple example which shows how to use the try...catch block. Let us put the following code in main.jsp –

```

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

```

```
<body>
  <%
    try {
      int i = 1;
      i = i / 0;
      out.println("The answer is " + i);
    }
    catch (Exception e) {
      out.println("An exception occurred: " + e.getMessage());
    }
  %>
</body>
</html>
```

Access the main.jsp, it should generate an output somewhat like the following –  
An exception occurred: / by zero

