## Introduction to JSP

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

**JSP pages are more advantageous than Servlet:**

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

**Features of JSP**

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

**JSP syntax**

Syntax available in JSP are following

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

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

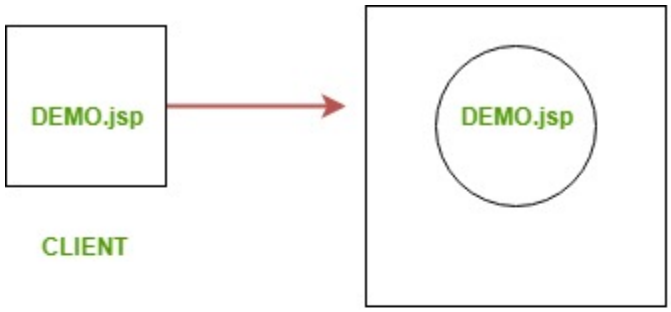
6. **Java Scriplets** :- It allows us to add any number of JAVA code, variables and expressions.

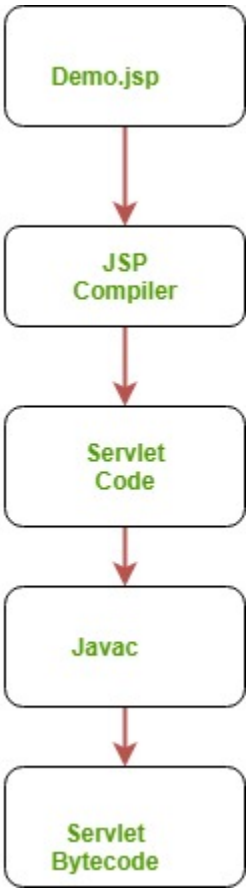
7. **Syntax:-**
8. `<% java code %>`
9. **JSP Expression** :- It evaluates and convert the expression to a string.
10. **Syntax:-**
11. `<%= expression %>`
12. **Example:-**
13. `<% num1 = num1+num2 %>`
14. **JAVA Comments** :- It contains the text that is added for information which has to be ignored.
15. **Syntax:-**
16. `<% -- JSP Comments %>`

## Process of Execution

Steps for Execution of JSP are following:-

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




**Example of Hello World**

We will make one .html file and .jsp file

**demo.jsp**

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
<title>Hello World - JSP tutorial</title>
</head>
<body>
    <%= "Hello World!" %>
</body>
</html>
```

## Advantages of using JSP

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

## Disadvantages of using JSP

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

## JSP Syntax and Semantics

It describes the main elements of a JSP page, namely scripting elements, directives, and actions. It also discusses additional elements like comments, implicit objects, custom tags. So let's start with JSP Syntax and Semantics.

# JSP Syntax and Semantics

As JSP deals with the writing of codes, a proper way to write that code becomes important. Thus we need to understand the syntax and semantics of a JSP code. To understand it better, we first need to know what syntax and semantics itself mean.

**Syntax –** Syntax is the coding structure to represent elements.
**Semantics –** It is the meaning of that element to the container i.e., what happens when that element is used. Thus we will discuss various elements that form the basis of JSP Syntax and Semantics

# Components of a JSP Page

1) JSP elements
2) Fixed template data
3) Or a combination of both

**JSP elements**

JSP containers will recognize these instructions. They tell about the generation of the code and how it needs to be operated. They have specific start and end tags for their recognition and identification by the web container.

As this data is static or fixe
This data won't change so
at the end, th

# JSP Elements

There are 3 most important elements in JSP

**1. JSP scripting elements**
1.1 Expressions
1.2 Scriptlets
1.3 Declarations
**2. Directives**
2.1page
2.2include
2.3taglib
**3. Actions**
Above mentioned elements form the most part of JSP code. In addition to this, various other elements are there like:

- Implicit objects
- JSP comments
- JSP custom tags

# 1. JSP Scripting Elements

## *1.1 Expressions*

They are a simple means for accessing the value of a Java variable. They can be an access point to other expressions and merging that value with HTML as well.

## Syntax of expressions is:

```
<%=expression/statement%>
```

This code gets written to the output stream of the response. Now, this expression or statement can be any valid Java statement. This code will convert to out.print() statement when an auto generated servlet forms until and unless the expression is convertible to string format.

## For Example:

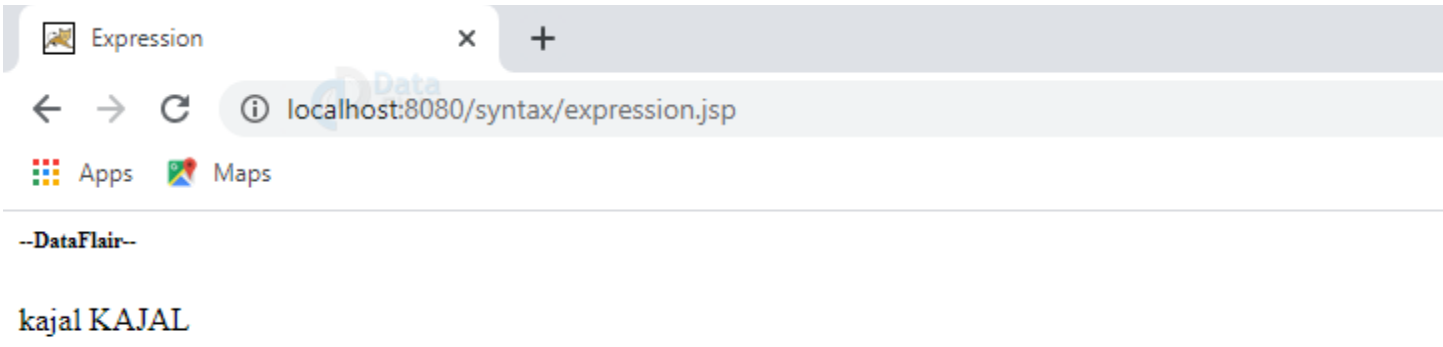
```
<html>

<head><title>Expression</title>




</head>

<body>
```

```
<h6>--DataFlair--</h6>




<% String s1="kajal"; %>


<% out.print(s1); %>
```

```
<% String s2=s1.toUpperCase(); %>


<% out.println( "  "+s2); %>




</body>


</html>
```

**Explanation:** Each line in the code is an expression. In this code we convert lower case to upper case.

## Output:




# *1.2 Scriptlets*

Scriptlets embed Java code in HTML for JSP code. It is like an expression, the only difference is that these are a set of statements written in Java language. It allows writing java code in it and is written between <%—-%> are called scriptlet tags. **Syntax of Scriptlet tag is as follows:**

```
<% statement; [statement;….]%>
```

Everything that is inside a scriptlet goes to the _jspService() of the servlet code for processing the request. In case a code has multiple scriptlets then they will append in the _jspService in an ordered fashion.

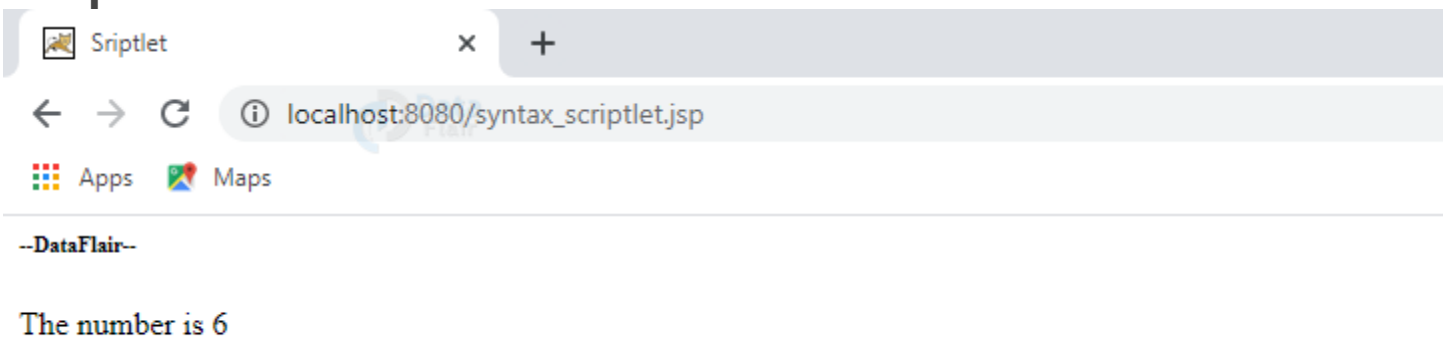
**For Example:**

```
<html>
```

```
<head><title>Scriptlet</title>




</head>

<body>

<h6>--DataFlair--</h6>
```

```
<% int a=1;

       int b=2;

       int c=3;

            out.println("The number is " + a*b*c);

%>

</body>

</html>
```

**Explanation:** In this code, we multiply three numbers and print the result in scriptlet tags.
**Output:**




# *1.3 Declarations*

Declarations too have java language statements. Yet there lies a major difference. This code doesn't go to _jspService(). This code incorporates into the source file that gets generated outside the _jspService method. **Syntax is:**

```
<%! statement,[statement,….] %>
```

Important points to note about declaration are that:

- They have no access to implicit objects(discussed further).

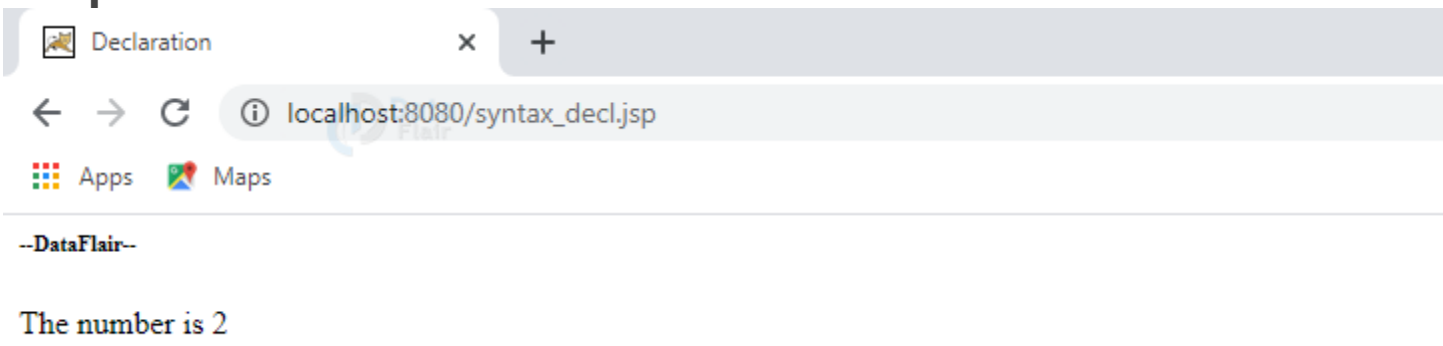
- They declare class/instance/variable/methods/inner classes.
- Declaration is inside the servlet class but remains outside the service method. As said earlier this code doesn't go to _jspService() method.
- If a method that is inside declaration wants to use requested object from scriptlet then it needs to pass the object as parameter.

## Example of declaration in JSP:

```
<html>

<head><title>Declaration</title>




</head>

<body>

<h6>--DataFlair--</h6>

<%! int num=1, n=0; %>

<% n=num+1;
```

```
        out.println("The number is " + n);


%>


</body>


</html>
```

**Explanation:** In this code, we have used declaration at code line 6, where we have declared the variables.
**Output:**




# 2. Directives

It is a type of instruction to the web container at the time of translation. Directives tell what code needs to be generated. It affects the compilation of the JSP page done by the container. **Syntax:**

```
<%@ directive name [attribute=”value”attribute=”value”…..]%>
```

***Note:*** They <%@……%> should lie in the same physical file.

**Classification of Directives under 3 categories is as follows:**

## *2.1 Page Directive*

It defines the attributes for the JSP page as a whole.

**Syntax:**

```
<%@ page[attribute=”value”attribute=”value”]%>
```

The attributes of the page directive are:

- language
- extends
- import
- session
- buffer
- autoflush
- isThreadsafe
- info
- isErrorpage
- errorpage

- contentType

**For Example:**

```
<%@ page language=”java” contentType=”text/.html”>
```

**Explanation:**
This code implies that coding of the page has been done in Java where the contentType of output is text or an HTML response. We won’t get an output here as directives are an offset to the JSP code only.

## *2.2 Include Directive*

It works similarly to the #include of the C preprocessor directory. It’s work is to merge the contents of another file into .jsp input stream at the translation phase. **Syntax:**

```
<%@ include file=”filename”%>
```

**For Example:**

```
<%@ include file=”header.html”>
```

```
<%@ include file=”sortmethod”>
```

**Explanation:** The “filename” can be the absolute name or the relative path name as shown in the example. There is no body coding here, only the use of include tag is shown so output is not there.

## *2.3 Taglib Directive*

With the use of tag library this directive makes the custom tags available in the current page. **Syntax can be shown as:**

```
<%@ taglib uri=”taglib URI”prefix=”tagPrefix”%>
```

The attributes of this directive are:

- tagLibrary URI
- tagPrefix

**The JSP directives section contains a detailed description of this topic.

# 3. Actions

Actions also known as standard actions unlike others are written in XML. These are also known as high level JSP

elements. They create, modify or use other objects. **Syntax:**

```
<tagname attr="value":….>….body…….</tagname>
```

Proper nesting is important here. For e.g.
<A><B>.....</B></A> is right whereas
<A><B>.....</A></B> is wrong.
Important tags in actions are namely:

- <jsp:usebean>
- <jsp:setProperty>
- <jsp:getProperty>
- <jsp:include>
- <jsp:forward>
- <jsp:param>
- <jsp:plugin>

**The article JSP Actions contains a detailed description of this topic.
Having discussed the three essential elements of the JSP page, let's discuss the periphery elements of a JSP page.

# JSP Comments

Comments contain statements and texts but are ignored. They simply describe the code so that the usability and understanding increases.

There are two types of comments in a JSP page. They are:
Firstly, JSPcomments that are written in the JSP code. JSP container ignores these comments. **Syntax:**

```
<%----Hidden JSP comment----%>
```

Secondly the HTML comments that are written in the XML or HTML code. Browser ignores these comments. **Syntax:**

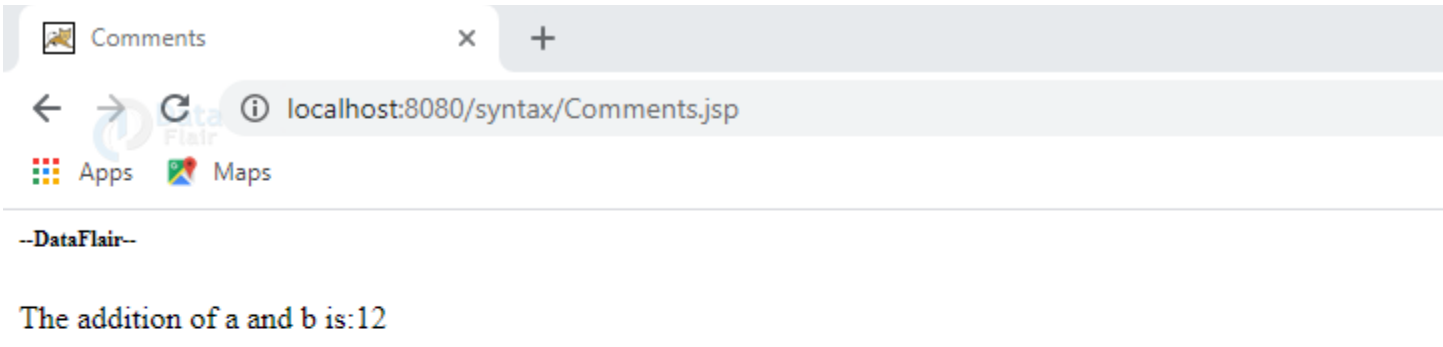
```
<!----included in generated HTML----!>
```

## Example:

```
<html>

<head><title>Comments</title>

<!---This is an HTML comment--->

<h6>--DataFlair--</h6>
```

```
</head>

<body>




<%---This is a JSP comment---%>

<% int a=1;

        int b=2;

        out.println("The addition of a and b is:" + a+b);

%>

</body>

</html>
```

**Explanation:** In the generated output you can compare, the code and result. JSP and HTML comments get ignored by the JSP container and browser respectively.
**Output:**




## Implicit Objects

As we know that only scriptlet, expression and HTML data goes to _jspService() and these variables are

implicitly available to all except declaration. Thus they are implicitly available; they don't need to be declared. Various implicit variables are request, response, pageContext, session, application, out, config, page, exception. They can also be created using tag libraries.

# Custom Tags

These tags can be created so that the functionality of JSP can be extended. They allow the encapsulation of functionality. This is thus made available to non-expert page authors.

## JAVA DEVELOPMENT MODEL

### Model 1 and Model 2 (MVC) Architecture



Before developing the web applications, we need to have idea about design models. There are two types of programming models (design models)

1. Model 1 Architecture
2. Model 2 (MVC) Architecture

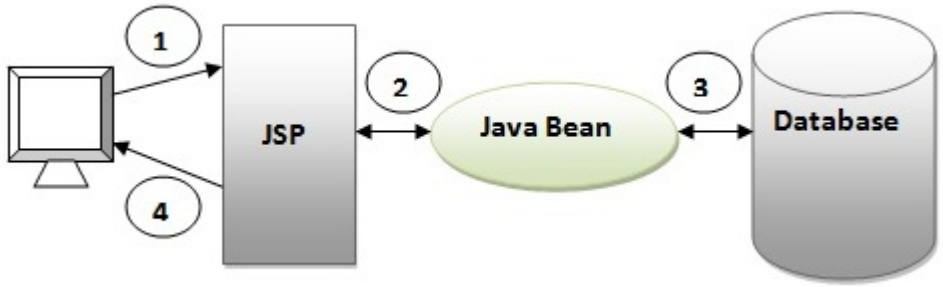
#### Model 1 Architecture

Servlet and JSP are the main technologies to develop the web applications.

**Servlet** was considered superior to CGI. Servlet technology doesn't create process, rather it creates thread to handle request. The advantage of creating thread over process is that it doesn't allocate separate memory area. Thus many subsequent requests can be easily handled by servlet.

**Problem in Servlet technology** Servlet needs to recompile if any designing code is modified. It doesn't provide separation of concern. Presentation and Business logic are mixed up.

**JSP** overcomes almost all the problems of Servlet. It provides better separation of concern, now presentation and business logic can be easily separated. You don't need to redeploy the application if JSP page is modified. JSP provides support to develop web application using JavaBean, custom tags and JSTL so that we can put the business logic separate from our JSP that will be easier to test and debug.




As you can see in the above figure, there is picture which show the flow of the model1 architecture.

1. Browser sends request for the JSP page
2. JSP accesses Java Bean and invokes business logic
3. Java Bean connects to the database and get/save data
4. Response is sent to the browser which is generated by JSP

## Advantage of Model 1 Architecture

- Easy and Quick to develop web application

## Disadvantage of Model 1 Architecture

- **Navigation control is decentralized** since every page contains the logic to determine the next page. If JSP page name is changed that is referred by other pages, we need to change it in all the pages that leads to the maintenance problem.
- **Time consuming** You need to spend more time to develop custom tags in JSP. So that we don't need to use scriptlet tag.
- **Hard to extend** It is better for small applications but not for large applications.

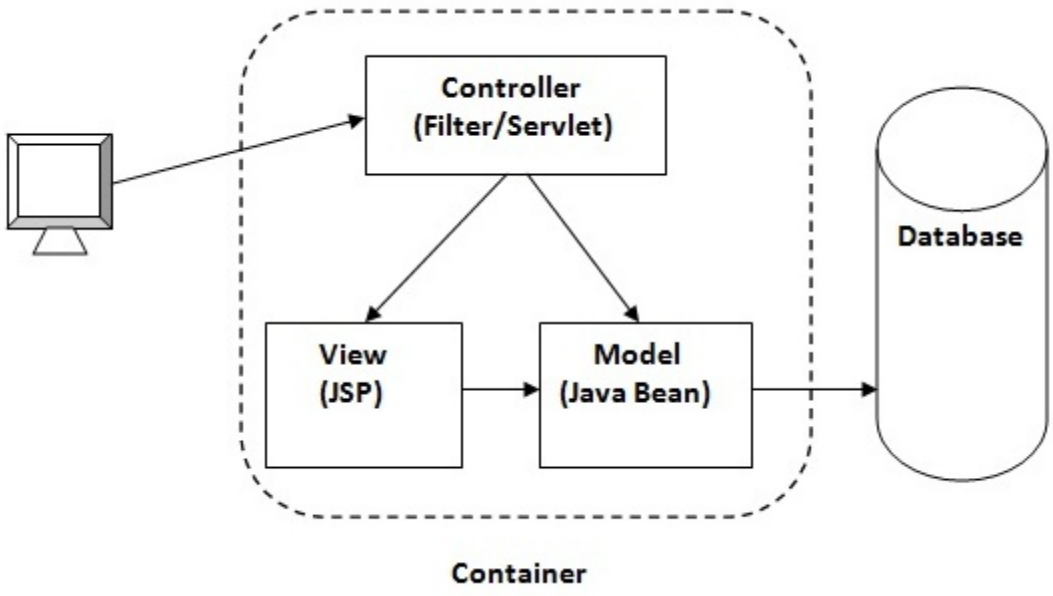
## Model 2 (MVC) Architecture

Model 2 is based on the MVC (Model View Controller) design pattern. The MVC design pattern consists of three modules model, view and controller.

**Model** The model represents the state (data) and business logic of the application.

**View** The view module is responsible to display data i.e. it represents the presentation.

**Controller** The controller module acts as an interface between view and model. It intercepts all the requests i.e. receives input and commands to Model / View to change accordingly.




## Advantage of Model 2 (MVC) Architecture

- **Navigation control is centralized** Now only controller contains the logic to determine the next page.
- **Easy to maintain**
- **Easy to extend**
- **Easy to test**
- **Better separation of concerns**

## Disadvantage of Model 2 (MVC) Architecture

- We need to write the controller code self. If we change the controller code, we need to recompile the class and redeploy the application.

Visit here to get the Example of MVC using Servlet and JSP.

## Solution of Model 2 Architecture: Configurable MVC Components

It uses the declarative approach for defining view components, request mapping etc. It resolves the problem of Model 2 architecture. The Struts framework provides the configurable MVC support. In struts 2, we define all the action classes and view components in struts.xml file.

COMPONENT OF JSP PAGE

# JSP Components

In this section we will discuss about the elements of JSP.

**Structure of JSP Page :**

JSPs are comprised of standard HTML tags and JSP tags. The structure of JavaServer pages are simple and easily handled by the servlet engine. In addition to HTML ,you can categorize JSPs as following -

- Directives
- Declarations
- Scriptlets
- Comments
- Expressions

**Directives :**

A directives tag always appears at the top of your JSP file. It is global definition sent to the JSP engine. Directives contain special processing instructions for the web container. You can import packages, define error handling pages or the session information of the JSP page. Directives are defined by using <%@ and %> tags.

**Syntax -**

<%@ directive attribute="value" %>

**Declarations :**

This tag is used for defining the functions and variables to be used in the JSP. This element of JSPs contains the java variables and methods which you can call in expression block of JSP page. Declarations are defined by using <%! and %> tags. Whatever you declare within these tags will be visible to the rest of the page.

**Syntax -**

<%! declaration(s) %>

**Scriptlets:**

In this tag we can insert any amount of valid java code and these codes are placed in _jspService method by the JSP engine. Scriptlets can be used anywhere in the page. Scriptlets are defined by using <% and %> tags.

**Syntax -**

<% Scriptlets%>

**Comments :**

Comments help in understanding what is actually code doing. JSPs provides two types of comments for putting comment in your page. First type of comment is for output comment which is appeared in the output stream on the browser. It is written by using the <!-- and --> tags.

**Syntax -**

<!-- comment text -->

Second type of comment is not delivered to the browser. It is written by using the <%-- and --%> tags.

**Syntax -**

<%-- comment text --%>

**Expressions :**

Expressions in JSPs is used to output any data on the generated page. These data are automatically converted to string and printed on the output stream. It is an instruction to the web container for executing the code with in the expression and replace it with the resultant output content. For writing expression in JSP, you can use <%= and %> tags.

**Syntax -**

<%= expression %>

EXPRESSION,SCRIPLET AND DECLARATION

# JSP Declaration Tag

The **JSP declaration tag** is used *to declare fields and methods*.

The code written inside the jsp declaration tag is placed outside the service() method of auto generated servlet.

So it doesn't get memory at each request.

*Syntax of JSP declaration tag*

The syntax of the declaration tag is as follows:

```
1. <%!  field or method declaration %>
```

## Difference between JSP Scriptlet tag and Declaration tag

| Jsp Scriptlet Tag | Jsp Declaration Tag |
|---|---|
| The jsp scriptlet tag can only declare variables not methods. | The jsp declaration tag can declare variables as well as methods. |
| The declaration of scriptlet tag is placed inside the _jspService() method. | The declaration of jsp declaration tag is placed outside the _jspService() method. |

## Example of JSP declaration tag that declares field

In this example of JSP declaration tag, we are declaring the field and printing the value of the declared field using the jsp expression tag.

index.jsp

```
1. <html>
2. <body>
3. <%! int data=50; %>
4. <%= "Value of the variable is:"+data %>
5. </body>
6. </html>
```

## Example of JSP declaration tag that declares method

In this example of JSP declaration tag, we are defining the method which returns the cube of given number and calling this method from the jsp expression tag. But we can also use jsp scriptlet tag to call the declared method.

index.jsp

```
1. <html>
2. <body>
3. <%!
4. int cube(int n){
5. return n*n*n*;
6. }
7. %>
8. <%= "Cube of 3 is:"+cube(3) %>
```

9. **</body>**
10. **</html>**

CREATING SIMPLE JSP PAGE

**JSP** technology is used to create web application just like Servlet technology. It can be thought of as an extension to Servlet because it provides more functionality than servlet such as expression language, JSTL, etc.

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

## Advantages of JSP over Servlet

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

*1) Extension to Servlet*

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

*2) Easy to maintain*

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

*3) Fast Development: No need to recompile and redeploy*

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

*4) Less code than Servlet*

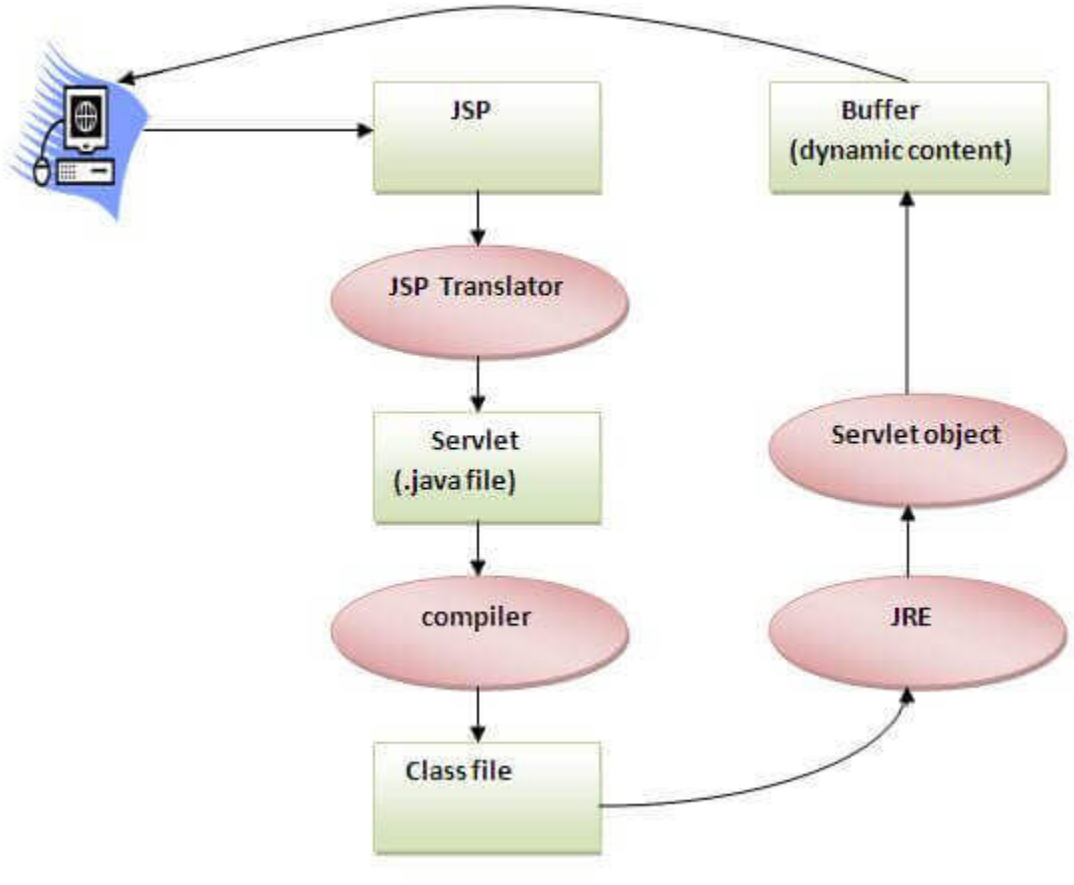
In JSP, we can use many tags such as action tags, JSTL, custom tags, etc. that reduces the code. Moreover, we can use EL, implicit objects, etc.

## The Lifecycle of a JSP Page

The JSP pages follow these phases:

- Translation of JSP Page
- Compilation of JSP Page
- Classloading (the classloader loads class file)
- Instantiation (Object of the Generated Servlet is created).

- Initialization ( the container invokes jspInit() method).
- Request processing ( the container invokes _jspService() method).
- Destroy ( the container invokes jspDestroy() method).




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

## Creating a simple JSP Page

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

**index.jsp**

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

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

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

## How to run a simple JSP Page?

Follow the following steps to execute this JSP page:

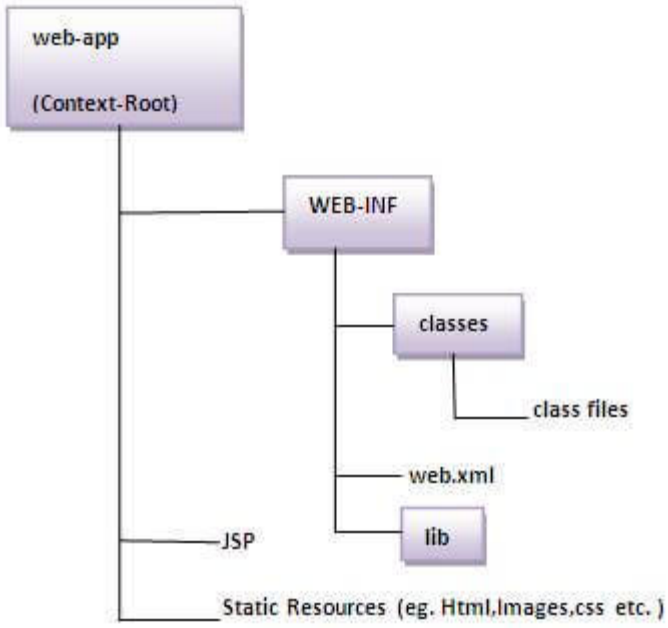
- Start the server
- Put the JSP file in a folder and deploy on the server
- Visit the browser by the URL http://localhost:portno/contextRoot/jspfile, for example, http://localhost:8888/myapplication/index.jsp

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

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

## The Directory structure of JSP

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




SESSION TRACKING

## Maintaining Session Between Web Client And Server

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

### Cookies

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

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

### Hidden Form Fields

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

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

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

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

### URL Rewriting

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

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

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

## The session Object

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

- a one page request or
- visit to a website or
- store information about that user

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

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

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

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

| S.No. | Method & Description |
|---|---|
| 1 | **public Object getAttribute(String name)**<br>This method returns the object bound with the specified name in this session, or null if no object is bound under the name. |
| 2 | **public Enumeration getAttributeNames()**<br>This method returns an Enumeration of String objects containing the names of all the objects bound to this session. |
| 3 | **public long getCreationTime()**<br>This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT. |
| 4 | **public String getId()**<br>This method returns a string containing the unique identifier assigned to this session. |
| 5 | **public long getLastAccessedTime()**<br>This method returns the last time the client sent a request associated with the this session, as the number of milliseconds since midnight January 1, 1970 GMT. |
| 6 | **public int getMaxInactiveInterval()**<br>This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses. |

| | |
|---|---|
| 7 | **public void invalidate()**<br>This method invalidates this session and unbinds any objects bound to it. |
| 8 | **public boolean isNew()**<br>This method returns true if the client does not yet know about the session or if the client chooses not to join the session. |
| 9 | **public void removeAttribute(String name)**<br>This method removes the object bound with the specified name from this session. |
| 10 | **public void setAttribute(String name, Object value)**<br>This method binds an object to this session, using the name specified. |
| 11 | **public void setMaxInactiveInterval(int interval)**<br>This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session. |

## Session Tracking Example

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

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

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

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

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

<html>
   <head>
      <title>Session Tracking</title>
   </head>
```

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

      <table border = "1" align = "center">
         <tr bgcolor = "#949494">
            <th>Session info</th>
            <th>Value</th>
         </tr>
         <tr>
            <td>id</td>
            <td><% out.print( session.getId()); %></td>
         </tr>
         <tr>
            <td>Creation Time</td>
            <td><% out.print(createTime); %></td>
         </tr>
         <tr>
            <td>Time of Last Access</td>
            <td><% out.print(lastAccessTime); %></td>
         </tr>
         <tr>
            <td>User ID</td>
            <td><% out.print(userID); %></td>
         </tr>
         <tr>
            <td>Number of visits</td>
            <td><% out.print(visitCount); %></td>
         </tr>
      </table>

   </body>
</html>
```

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.

# FORM EDITING IN JSP

# Registration Form in JSP

For creating registration form, you must have a table in the database. You can write the database logic in JSP file, but separating it from the JSP page is better approach. Here, we are going to use DAO, Factory Method, DTO and Singletion design patterns. There are many files:

- **index.jsp** for getting the values from the user
- **User.java**, a bean class that have properties and setter and getter methods.
- **process.jsp**, a jsp file that processes the request and calls the methods

- **Provider.java**, an interface that contains many constants like DRIVER_CLASS, CONNECTION_URL, USERNAME and PASSWORD
- **ConnectionProvider.java**, a class that returns an object of Connection. It uses the Singleton and factory method design pattern.
- **RegisterDao.java**, a DAO class that is responsible to get access to the database

## Example of Registration Form in JSP

In this example, we are using the Oracle10g database to connect with the database. Let's first create the table in the Oracle database:

```
1. CREATE TABLE  "USER432"
2.    (    "NAME" VARCHAR2(4000),
3.     "EMAIL" VARCHAR2(4000),
4.     "PASS" VARCHAR2(4000)
5.    )
6. /
```

We have created the table named user432 here.

*index.jsp*

We are having only three fields here, to make the concept clear and simplify the flow of the application. You can have other fields also like country, hobby etc. according to your requirement.

```
1. <form action="process.jsp">
2. <input type="text" name="uname" value="Name..." onclick="this.value=""/><br/>
3. <input type="text" name="uemail"  value="Email ID..." onclick="this.value=""/><br/>
4. <input type="password" name="upass"  value="Password..." onclick="this.value=""/><br/>
5. <input type="submit" value="register"/>
6. </form>
```

*process.jsp*

This jsp file contains all the incoming values to an object of bean class which is passed as an argument in the register method of the RegisterDao class.

```
1. <%@page import="bean.RegisterDao"%>
2. <jsp:useBean id="obj" class="bean.User"/>
3. _
4. <jsp:setProperty property="*" name="obj"/>
5. _
6. <%
```

```
int status=RegisterDao.register(obj);
if(status>0)
out.print("You are successfully registered");
_
%>
```

*User.java*

It is the bean class that have 3 properties uname, uemail and upass with its setter and getter methods.

```
package bean;
_
public class User {
private String uname,upass,uemail;
_
public String getUname() {
    return uname;
}
_
public void setUname(String uname) {
    this.uname = uname;
}
_
public String getUpass() {
    return upass;
}
_
public void setUpass(String upass) {
    this.upass = upass;
}
_
public String getUemail() {
    return uemail;
}
_
public void setUemail(String uemail) {
    this.uemail = uemail;
}
_
}
```

*Provider.java*

This interface contains four constants that can vary from database to database.

```
package bean;
```

```
2.
3.  public interface Provider {
4.  String DRIVER="oracle.jdbc.driver.OracleDriver";
5.  String CONNECTION_URL="jdbc:oracle:thin:@localhost:1521:xe";
6.  String USERNAME="system";
7.  String PASSWORD="oracle";
8.
9.  }
```

*ConnectionProvider.java*

This class is responsible to return the object of Connection. Here, driver class is loaded only once and connection object gets memory only once.



```
1.  package bean;
2.  import java.sql.*;
3.  import static bean.Provider.*;
4.
5.  public class ConnectionProvider {
6.  private static Connection con=null;
7.  static{
8.  try{
9.  Class.forName(DRIVER);
10. con=DriverManager.getConnection(CONNECTION_URL,USERNAME,PASSWORD);
11. }catch(Exception e){}
12. }
13.
14. public static Connection getCon(){
15.     return con;
16. }
17.
18. }
```

*RegisterDao.java*

This class inserts the values of the bean component into the database.

```
1.  package bean;
2.
3.  import java.sql.*;
4.
5.  public class RegisterDao {
6.
7.  public static int register(User u){
```

```
8.  int status=0;
9.  try{
10. Connection con=ConnectionProvider.getCon();
11. PreparedStatement ps=con.prepareStatement("insert into user432 values(?,?,?)");
12. ps.setString(1,u.getUname());
13. ps.setString(2,u.getUemail());
14. ps.setString(3,u.getUpass());
15. 
16. status=ps.executeUpdate();
17. }catch(Exception e){}
18. 
19. return status;
20. }
21. 
22. }
```

# Login and Logout Example in JSP

1. Login and Logout Example in JSP
2. Example of Login Form in JSP

In this example of creating login form, we have used the DAO (Data Access Object), Factory method and DTO (Data Transfer Object) design patterns. There are many files:

- **index.jsp** it provides three links for login, logout and profile
- **login.jsp** for getting the values from the user
- **loginprocess.jsp**, a jsp file that processes the request and calls the methods.
- **LoginBean.java**, a bean class that have properties and setter and getter methods.
- **Provider.java**, an interface that contains many constants like DRIVER_CLASS, CONNECTION_URL, USERNAME and PASSWORD
- **ConnectionProvider.java**, a class that is responsible to return the object of Connection. It uses the Singleton and factory method design pattern.
- **LoginDao.java**, a DAO class that verifies the emailId and password from the database.
- **logout.jsp** it invalidates the session.
- **profile.jsp** it provides simple message if user is logged in, otherwise forwards the request to the login.jsp page.

In this example, we are using the Oracle10g database to match the emailId and password with the database. The table name is user432 which have many fields like name, email, pass etc. You may use this query to create the table:

```
1. CREATE TABLE  "USER432"
2.    (    "NAME" VARCHAR2(4000),
```

```
3.      "EMAIL" VARCHAR2(4000),
4.      "PASS" VARCHAR2(4000)
5.     )
6.  /
```

We assume that there are many records in this table.

*index.jsp*

It simply provides three links for login, logout and profile.

```
1.  <a href="login.jsp">login</a>|
2.  <a href="logout.jsp">logout</a>|
3.  <a href="profile.jsp">profile</a>
```

*login.jsp*

This file creates a login form for two input fields name and password. It is the simple login form, you can change it for better look and feel. We are focusing on the concept only.

```
1.  <%@ include file="index.jsp" %>
2.  <hr/>
3.
4.  <h3>Login Form</h3>
5.  <%
6.  String profile_msg=(String)request.getAttribute("profile_msg");
7.  if(profile_msg!=null){
8.  out.print(profile_msg);
9.  }
10. String login_msg=(String)request.getAttribute("login_msg");
11. if(login_msg!=null){
12. out.print(login_msg);
13. }
14.  %>
15.  <br/>
16. <form action="loginprocess.jsp" method="post">
17. Email:<input type="text" name="email"/><br/><br/>
18. Password:<input type="password" name="password"/><br/><br/>
19. <input type="submit" value="login"/>"
20. </form>
```

*loginprocess.jsp*

This jsp file contains all the incoming values to an object of bean class which is passed as an argument in the validate method of the LoginDao class. If emailid and password is correct, it displays a message you are successfully logged in! and maintains the session so that we may recognize the user.

```
1.  <%@page import="bean.LoginDao"%>
2.  <jsp:useBean id="obj" class="bean.LoginBean"/>
3.  
4.  <jsp:setProperty property="*" name="obj"/>
5.  
6.  <%
7.  boolean status=LoginDao.validate(obj);
8.  if(status){
9.  out.println("You r successfully logged in");
10. session.setAttribute("session","TRUE");
11. }
12. else
13. {
14. out.print("Sorry, email or password error");
15. %>
16. <jsp:include page="index.jsp"></jsp:include>
17. <%
18. }
19. %>
```

*LoginBean.java*

It is the bean class that have 2 properties email and pass with its setter and getter methods.

```
1.  package bean;
2.  
3.  public class LoginBean {
4.  private String email,pass;
5.  
6.  public String getEmail() {
7.     return email;
8.  }
9.  
10. public void setEmail(String email) {
11.    this.email = email;
12. }
13. 
14. public String getPass() {
15.    return pass;
16. }
17. 
18. public void setPass(String pass) {
```

```
19.     this.pass = pass;
20.  }
21.  
22.  
23.  }
```

*Provider.java*

This interface contains four constants that may differ from database to database.

```
1.  package bean;
2.  
3.  public interface Provider {
4.  String DRIVER="oracle.jdbc.driver.OracleDriver";
5.  String CONNECTION_URL="jdbc:oracle:thin:@localhost:1521:xe";
6.  String USERNAME="system";
7.  String PASSWORD="oracle";
8.  
9.  }
```

*ConnectionProvider.java*

This class provides a factory method that returns the object of Connection. Here, driver class is loaded only once and connection object gets memory only once because it is static.

```
1.  package bean;
2.  import java.sql.*;
3.  import static bean.Provider.*;
4.  
5.  public class ConnectionProvider {
6.  private static Connection con=null;
7.  static{
8.  try{
9.  Class.forName(DRIVER);
10. con=DriverManager.getConnection(CONNECTION_URL,USERNAME,PASSWORD);
11. }catch(Exception e){}
12. }
13. 
14. public static Connection getCon(){
15.     return con;
16. }
17. 
18. }
```

*LoginDao.java*

This class varifies the emailid and password.

```
1.  package bean;
2.  import java.sql.*;
3.  public class LoginDao {
4.  
5.  public static boolean validate(LoginBean bean){
6.  boolean status=false;
7.  try{
8.  Connection con=ConnectionProvider.getCon();
9.  
10. PreparedStatement ps=con.prepareStatement(
11.    "select * from user432 where email=? and pass=?");
12. 
13. ps.setString(1,bean.getEmail());
14. ps.setString(2, bean.getPass());
15. 
16. ResultSet rs=ps.executeQuery();
17. status=rs.next();
18. 
19. }catch(Exception e){}
20. 
21. return status;
22. 
23. }
24. }
```

# Uploading file to the server using JSP



There are many ways to upload the file to the server. One of the way is by the MultipartRequest class. For using this class you need to have the cos.jar file. In this example, we are providing the cos.jar file alongwith the code.

## MultipartRequest class

It is a utility class to handle the multipart/form-data request. There are many constructors defined in the MultipartRequest class.

### Commonly used Constructors of MultipartRequest class

- **MultipartRequest(HttpServletRequest request, String saveDirectory)** uploads the file upto 1MB.
- **MultipartRequest(HttpServletRequest request, String saveDirectory, int maxPostSize)** uploads the file upto specified post size.

- **MultipartRequest(HttpServletRequest request, String saveDirectory, int maxPostSize, String encoding)** uploads the file upto specified post size with given encoding.

## Example of File Upload in JSP

In this example, we are creating two files only, index.jsp and fileupload.jsp.

*index.jsp*

To upload the file to the server, there are two requirements:

1. You must use the post request.
2. encodeType should be multipart/form-data that gives information to the server that you are going to upload the file.

```
1. <form action="upload.jsp" method="post" enctype="multipart/form-data">
2. Select File:<input type="file" name="fname"/><br/>
3. <input type="image" src="MainUpload.png"/>
4. </form>
```

*upload.jsp*

We are uploading the incoming file to the location d:/new, you can specify your location here.

```
1. <%@ page import="com.oreilly.servlet.MultipartRequest" %>
2. <%
3. MultipartRequest m = new MultipartRequest(request, "d:/new");
4. out.print("successfully uploaded");
5. _
6. %>
```

JSP APPLICATION

# Introduction to Java Database Connectivity

Java Database Connectivity **(**JDBC**) is an application programming interface (API) for the programming language Java,**

**which defines how a client may access a database. It is a Java-based data access technology used for Java database**

**connectivity. It is part of the Java Standard Edition platform, from Oracle Corporation. It provides methods to query and update data in a database, and is oriented toward relational databases. A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the Java virtual**

JDBC ('Java Database Connectivity') allows multiple implementations to exist and be used by the same application. The API provides a mechanism for dynamically loading the correct Java packages and registering them with the JDBC Driver Manager. The Driver Manager is used as a connection factory for creating JDBC connections.

JDBC connections support creating and executing statements. These may be update statements such as SQL's CREATE, INSERT, UPDATE and DELETE, or they may be query statements such as SELECT. Additionally, stored procedures may be invoked through a JDBC connection. JDBC represents statements using one of the following classes:

- `Statement` – the statement is sent to the database server each and every time.
- `PreparedStatement` – the statement is cached and then the execution path is pre-determined on the database server allowing it to be executed multiple times in an efficient manner.
- `CallableStatement` – used for executing stored procedures on the database.

Update statements such as INSERT, UPDATE and DELETE return an update count that indicates how many rows were affected in the database. These statements do not return any other information.

Query statements return a JDBC row result set. The row result set is used to walk over the result set. Individual columns in a row are retrieved either by name or by column number. There may be any number of rows in the result set. The row result set has metadata that describes the names of the columns and their types.

There is an extension to the basic JDBC API in the `javax.sql`.

JDBC connections are often managed via a connection pool rather than obtained directly from the driver.

**Host database types which Java can convert to with a function**

| Oracle Datatype | `setXXX()` Methods |
|---|---|
| CHAR | `setString()` |
| VARCHAR2 | `setString()` |
| NUMBER | `setBigDecimal()` |
| | `setBoolean()` |
| | `setByte()` |
| | `setShort()` |
| | `setInt()` |
| | `setLong()` |
| | `setFloat()` |
| | `setDouble()` |
| INTEGER | `setInt()` |
| FLOAT | `setDouble()` |

| | |
|---|---|
| CLOB | `setClob()` |
| BLOB | `setBlob()` |
| RAW | `setBytes()` |
| LONGRAW | `setBytes()` |
| DATE | `setDate()` |
| | `setTime()` |
| | `setTimestamp()` |

## Examples[edit]

When a Java application needs a database connection, one of the `DriverManager.getConnection()` methods is used to create a JDBC connection. The URL used is dependent upon the particular database and JDBC driver. It will always begin with the "jdbc:" protocol, but the rest is up to the particular vendor.

```
Connection conn = DriverManager.getConnection(
     "jdbc:somejdbcvendor:other data needed by some jdbc vendor",
     "myLogin",
     "myPassword");
try {
     /* you use the connection here */
} finally {
    //It's important to close the connection when you are done with it
    try {
        conn.close();
    } catch (Throwable e) { /* Propagate the original exception
                               instead of this one that you want just logged
*/
        logger.warn("Could not close JDBC Connection",e);
    }
}
```

Starting from Java SE 7 you can use Java's try-with-resources statement to make the above code simpler:

```
try (Connection conn = DriverManager.getConnection(
     "jdbc:somejdbcvendor:other data needed by some jdbc vendor",
     "myLogin",
     "myPassword")) {
     /* you use the connection here */
}  // the VM will take care of closing the connection
```

Once a connection is established, a statement can be created.

```
try (Statement stmt = conn.createStatement()) {
    stmt.executeUpdate("INSERT INTO MyTable(name) VALUES ('my name')");
```

```
}
```

Note that Connections, Statements, and ResultSets often tie up operating system resources such as sockets or file descriptors. In the case of Connections to remote database servers, further resources are tied up on the server, e.g., cursors for currently open ResultSets. It is vital to `close()` any JDBC object as soon as it has played its part; garbage collection should not be relied upon. The above try-with-resources construct is a code pattern that obviates this.

Data is retrieved from the database using a database query mechanism. The example below shows creating a statement and executing a query.

```
try (Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM MyTable")
) {
    while (rs.next()) {
        int numColumns = rs.getMetaData().getColumnCount();
        for (int i = 1; i <= numColumns; i++) {
           // Column numbers start at 1.
           // Also there are many methods on the result set to return
           //  the column as a particular type. Refer to the Sun
documentation
           //  for the list of valid conversions.
           System.out.println( "COLUMN " + i + " = " + rs.getObject(i));
        }
    }
}
```

An example of a `PreparedStatement` query, using `conn` and class from first example.

```
try (PreparedStatement ps =
    conn.prepareStatement("SELECT i.*, j.* FROM Omega i, Zappa j WHERE i.name
= ? AND j.num = ?")
) {
    // In the SQL statement being prepared, each question mark is a
placeholder
    // that must be replaced with a value you provide through a "set" method
invocation.
    // The following two method calls replace the two placeholders; the first
is
    // replaced by a string value, and the second by an integer value.
    ps.setString(1, "Poor Yorick");
    ps.setInt(2, 8008);

    // The ResultSet, rs, conveys the result of executing the SQL statement.
    // Each time you call rs.next(), an internal row pointer, or cursor,
    // is advanced to the next row of the result.  The cursor initially is
    // positioned before the first row.
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
            int numColumns = rs.getMetaData().getColumnCount();
            for (int i = 1; i <= numColumns; i++) {
                // Column numbers start at 1.
                // Also there are many methods on the result set to return
                // the column as a particular type. Refer to the Sun
documentation
```

```
                // for the list of valid conversions.
                System.out.println("COLUMN " + i + " = " + rs.getObject(i));
            } // for
        } // while
    } // try
} // try
```

If a database operation fails, JDBC raises an `SQLException`. There is typically very little one can do to recover from such an error, apart from logging it with as much detail as possible. It is recommended that the SQLException be translated into an application domain exception (an unchecked one) that eventually results in a transaction rollback and a notification to the user.

An example of a database transaction:

```
boolean autoCommitDefault = conn.getAutoCommit();
try {
    conn.setAutoCommit(false);

    /* You execute statements against conn here transactionally */

    conn.commit();
} catch (Throwable e) {
    try { conn.rollback(); } catch (Throwable e) { logger.warn("Could not
rollback transaction", e); }
    throw e;
} finally {
    try { conn.setAutoCommit(autoCommitDefault); } catch (Throwable e) {
logger.warn("Could not restore AutoCommit setting",e); }
}
```

For an example of a `CallableStatement` (to call stored procedures in the database), see the JDBC API Guide documentation.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;

public class Mydb1 {
   static String URL = "jdbc:mysql://localhost/mydb";

   public static void main(String[] args) {
      try {
        Class.forName("com.mysql.jdbc.Driver");

        Connection conn = DriverManager.getConnection(URL, "root", "root");
        Statement stmt = conn.createStatement();

        String sql = "INSERT INTO emp1 VALUES ('pctb5361', 'kiril', 'john',
968666668)";
        stmt.executeUpdate(sql);

        System.out.println("Inserted records into the table...");
      } catch (Exception e) {
        e.printStackTrace();
      }
```

```
    }
}
```

# JDBC Driver

1. JDBC Drivers
1. JDBC-ODBC bridge driver
2. Native-API driver
3. Network Protocol driver
4. Thin driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

## 1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

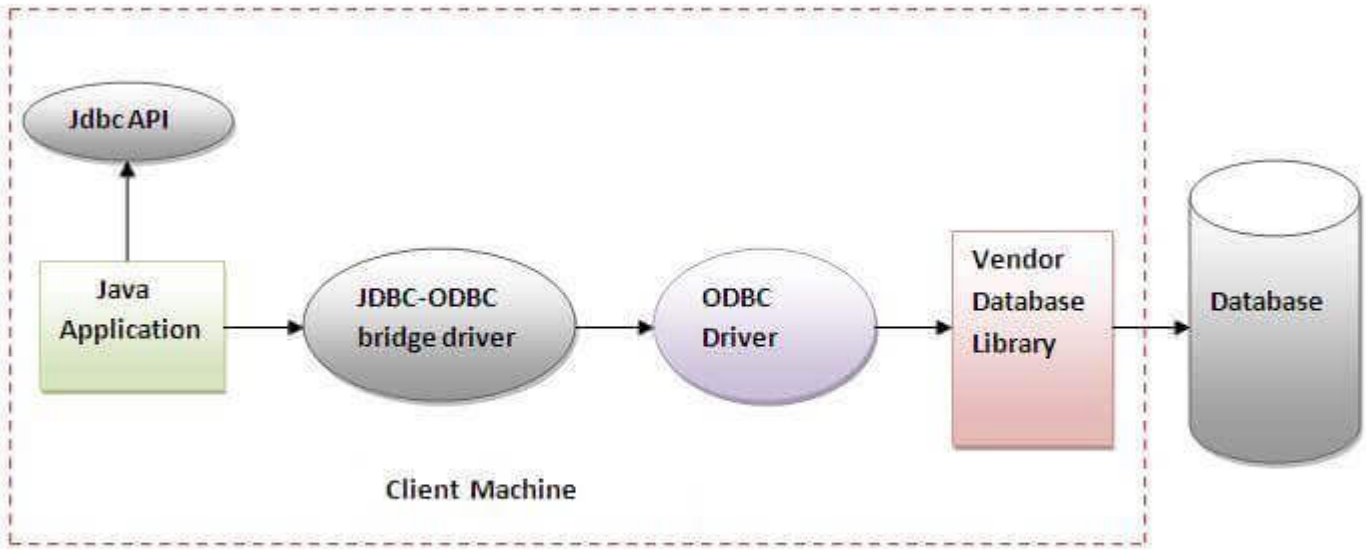



Figure- JDBC-ODBC Bridge Driver

*In Java 8, the JDBC-ODBC Bridge has been removed.*

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

## 2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

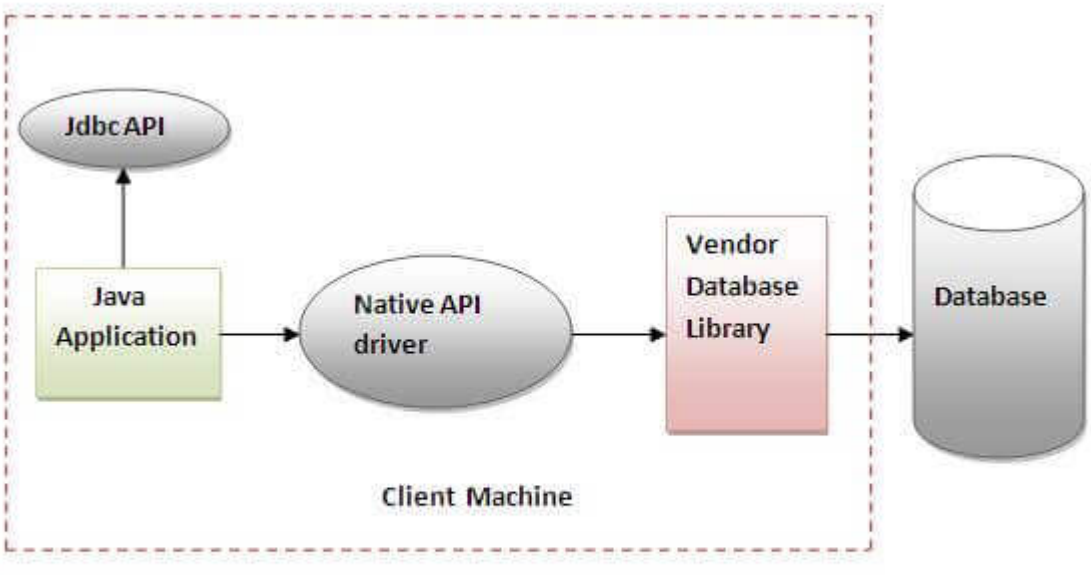



Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.

- The Vendor client library needs to be installed on client machine.

## 3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

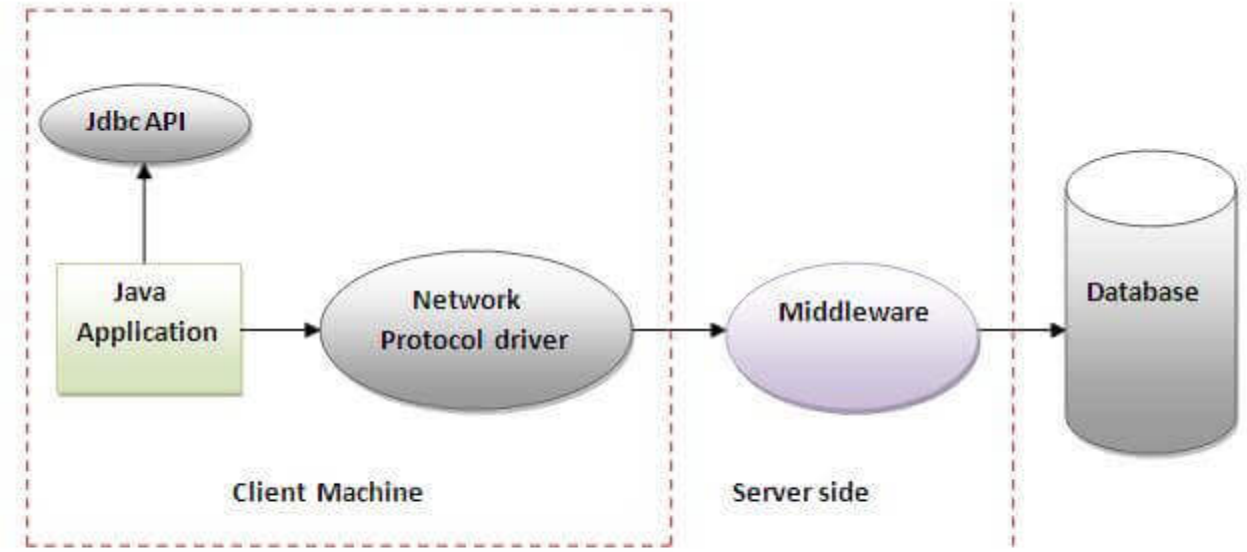



Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

## 4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known

as thin driver. It is fully written in Java language.

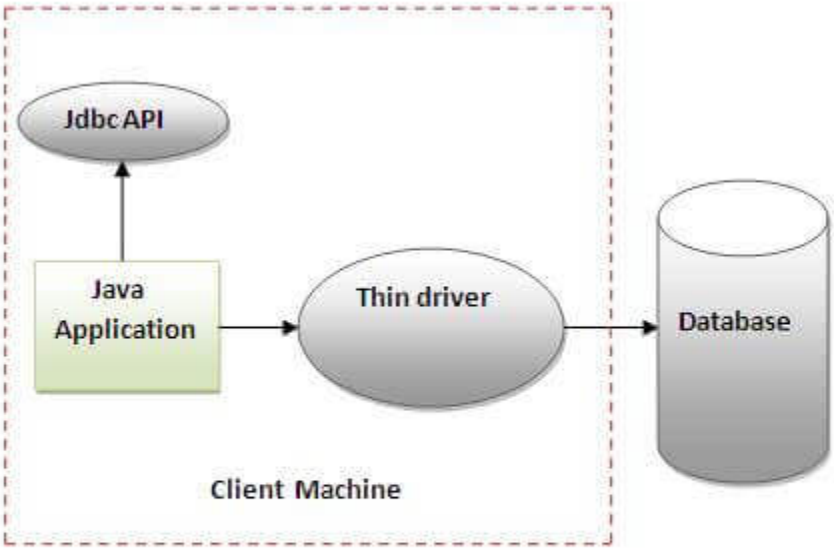



Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

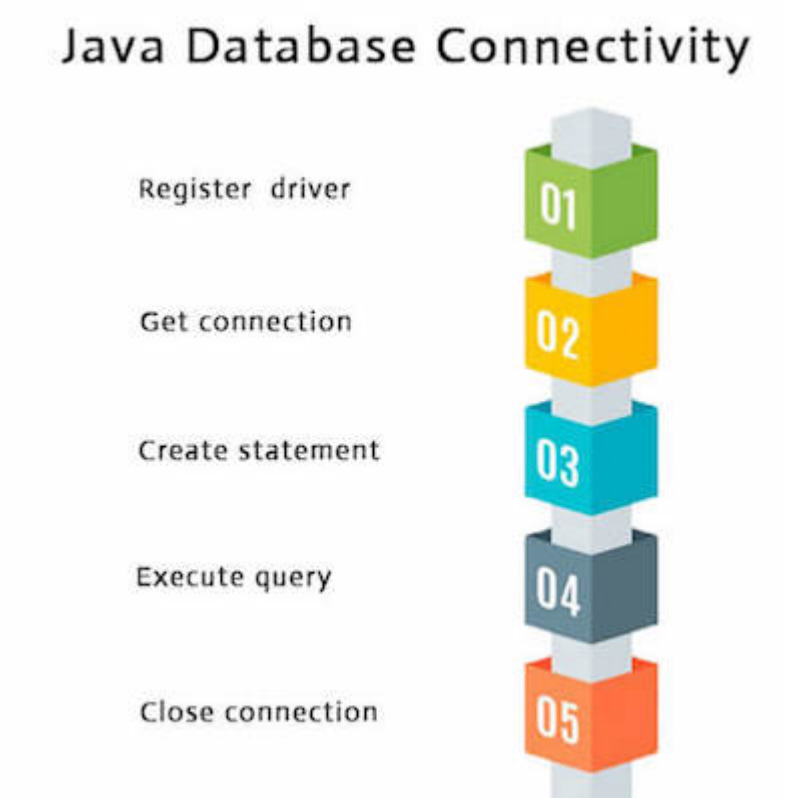
# Java Database Connectivity with 5 Steps

1. 5 Steps to connect to the database in java
1. Register the driver class
2. Create the connection object
3. Create the Statement object
4. Execute the query
5. Close the connection object

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class

- Create connection
- Create statement
- Execute queries
- Close connection




## 1) Register the driver class

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

### Syntax of forName() method

```
1. public static void forName(String className)throws ClassNotFoundException
```

> *Note: Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vender's Jar in the classpath, and then JDBC driver manager can detect and load the driver automatically.*

## Example to register the OracleDriver class

Here, Java program is loading oracle driver to esteblish database connection.

```
1. Class.forName("oracle.jdbc.driver.OracleDriver");
```

## 2) Create the connection object

The **getConnection()** method of DriverManager class is used to establish connection with the database.

### Syntax of getConnection() method

1) **public static** Connection getConnection(String url)**throws** SQLException
2) **public static** Connection getConnection(String url,String name,String password)
3) **throws** SQLException

## Example to establish connection with the Oracle database

```
1. Connection con=DriverManager.getConnection(
2. "jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

## 3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

### Syntax of createStatement() method

1) **public** Statement createStatement()**throws** SQLException

## Example to create the statement object

```
1. Statement stmt=con.createStatement();
```

## 4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

### Syntax of executeQuery() method

1) **public** ResultSet executeQuery(String sql)**throws** SQLException

## Example to execute query

```
1. ResultSet rs=stmt.executeQuery("select * from emp");
2. 
3. while(rs.next()){
4. System.out.println(rs.getInt(1)+" "+rs.getString(2));
5. }
```

## 5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

### Syntax of close() method

1. **public void** close()**throws** SQLException

## Example to close connection

1. con.close();

# Connecting to the database with driver

The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps −

- **Import JDBC Packages** − Add **import** statements to your Java program to import required classes in your Java code.
- **Register JDBC Driver** − This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- **Database URL Formulation** − This is to create a properly formatted address that points to the database to which you wish to connect.
- **Create Connection Object** − Finally, code a call to the *DriverManager* object's *getConnection( )* method to establish actual database connection.

## Import JDBC Packages

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code −

```
import java.sql.* ;  // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

## Register JDBC Driver

You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces.

You need to do this registration only once in your program. You can register a driver in one of two ways.

## Approach I - Class.forName()

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses Class.forName( ) to register the Oracle driver −

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

You can use **getInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows −

```
try {
```

```
   Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
catch(IllegalAccessException ex) {
   System.out.println("Error: access problem while loading!");
   System.exit(2);
catch(InstantiationException ex) {
   System.out.println("Error: unable to instantiate driver!");
   System.exit(3);
}
```

## Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver −

```
try {
   Driver myDriver = new oracle.jdbc.driver.OracleDriver();
   DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

## Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods −

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

| RDBMS | JDBC driver name | URL format |
|---|---|---|
| MySQL | com.mysql.jdbc.Driver | **jdbc:mysql://**hostname/ databaseName |
| ORACLE | oracle.jdbc.driver.OracleDriver | **jdbc:oracle:thin:@**hostname:port Number:databaseName |
| DB2 | COM.ibm.db2.jdbc.net.DB2Driver | **jdbc:db2:**hostname:port Number/databaseName |
| Sybase | com.sybase.jdbc.SybDriver | **jdbc:sybase:Tds:**hostname: port |

| | | |
|---|---|---|
| | | Number/databaseName |

All the highlighted part in URL format is static and you need to change only the remaining part as per your database setup.

## Create Connection Object

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object.

## Using a Database URL with a username and password

The most commonly used form of getConnection() requires you to pass a database URL, a *username*, and a *password* −

Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would be −

```
jdbc:oracle:thin:@amrood:1521:EMP
```

Now you have to call getConnection() method with appropriate username and password to get a **Connection** object as follows −

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

## Using Only a Database URL

A second form of the DriverManager.getConnection( ) method requires only a database URL −

```
DriverManager.getConnection(String url);
```

However, in this case, the database URL includes the username and password and has the following general form −

```
jdbc:oracle:driver:username/password@database
```

So, the above connection can be created as follows −

```
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

## Using a Database URL and a Properties Object

A third form of the DriverManager.getConnection( ) method requires a database URL and a Properties object −

```
DriverManager.getConnection(String url, Properties info);
```

A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples, use the following code −

```
import java.util.*;

String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );

Connection conn = DriverManager.getConnection(URL, info);
```

## Closing JDBC Connections

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object.

To ensure that a connection is closed, you could provide a 'finally' block in your code. A *finally* block always executes, regardless of an exception occurs or not.

To close the above opened connection, you should call close() method as follows −

```
conn.close();
```