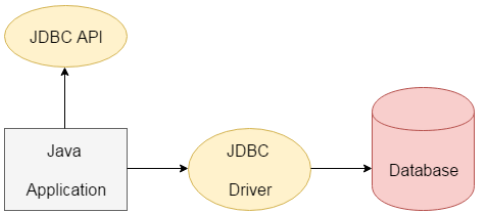
## JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver




We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

## JDBC 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 driv converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

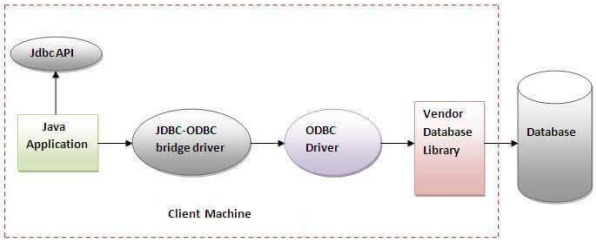



Figure- JDBC-ODBC Bridge Driver

*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 cal into native calls of the database API. It is not written entirely in java.

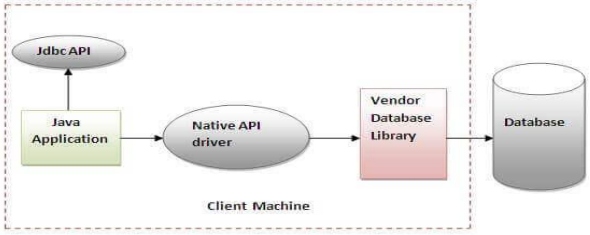



Figure- Native API Driver

**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.

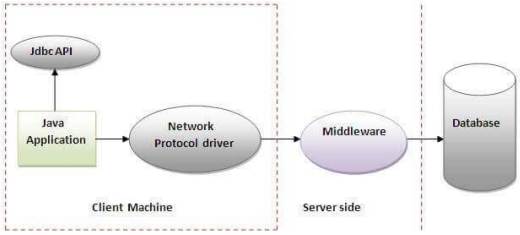



Figure- Network Protocol Driver

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

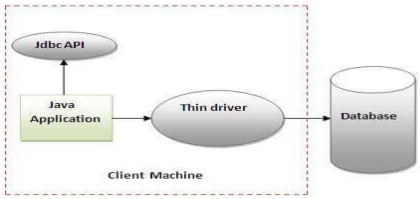



Figure- Thin Driver

**Advantage:**

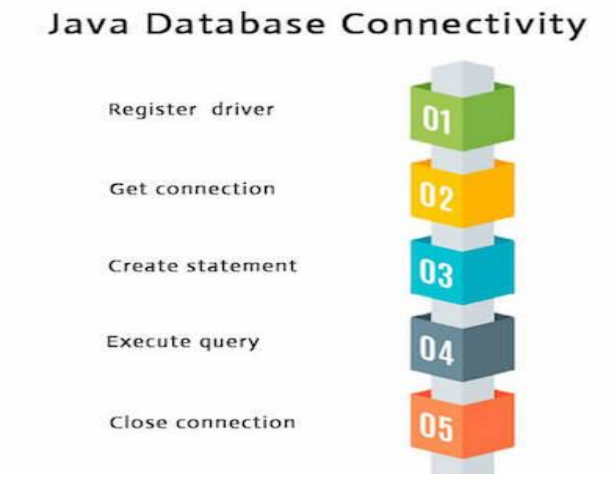
- Better performance than all other drivers.
- No software is required at client side or server side.

**Disadvantage:**

- Drivers depend on the Database.

**Java Database Connectivity**

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




**1) Register the driver class**

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

**Syntax of forName() method**

**public static void** forName(String className)**throws** ClassNotFoundException

**Example to register the OracleDriver class**

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

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) **throws** SQLException

**Example to establish connection with the Oracle database**

Connection con=DriverManager.getConnection(
"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 statemer is responsible to execute queries with the database.

**Syntax of createStatement() method**

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

**Example to create the statement object**

Statement stmt=con.createStatement();

**4) Execute the query**

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

**Syntax of executeQuery() method**

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

**Example to execute query**

```
ResultSet rs=stmt.executeQuery("select * from emp");

while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

**5) Close the connection object**

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

**Syntax of close() method**

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

**Example to close connection**

```
con.close();
```

**Statement interface**

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

**Commonly used methods of Statement interface:**

The important methods of Statement interface are as follows:

**1) public ResultSet executeQuery(String sql):** is used to execute SELECT query.

It returns the object of ResultSet.

**2) public int executeUpdate(String sql):** is used to execute specified query, it may be create, drop,

insert, update, delete etc.

**3) public boolean execute(String sql):** is used to execute queries that may return multiple results.

**4) public int[] executeBatch():** is used to execute batch of commands.

**Example of Statement interface**

Let's see the simple example of Statement interface to insert, update and delete the record.

```
import java.sql.*;

class FetchRecord{

public static void main(String args[])throws Exception{

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

Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","syst
em","oracle");

Statement stmt=con.createStatement();

  //stmt.executeUpdate("insert into emp765 values(33,'Irfan',50000)");

//int result=stmt.executeUpdate("update emp765 set name='Vimal',salary=10000 where i
d=33");

int result=stmt.executeUpdate("delete from emp765 where id=33");

System.out.println(result+" records affected");

con.close();

}}
```

**Creating and executing JDBC statements**

**CREATE**

```
package com.mycompany.labexperiments;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
public class LabExperiments {

    public static void main(String[] args) {
        String jdbcUrl = "jdbc:mysql://localhost:3306/mysql";
        String username = "root";
```

```
        String password = "root";
        Connection con;

        try {
           con = DriverManager.getConnection(jdbcUrl, username, password);
           Statement stmt;
           String str;
                  str =  "create table Alex " +
                                "(Name VARCHAR(32)," +
                                "Age INTEGER," +
                                "Salary FLOAT)";
                  stmt = con.createStatement();
                  stmt.executeUpdate(str);
                  stmt.close();
                  con.close();
     }
     catch(SQLException vc) {
        System.out.println(vc);
     }
     try
     {
                   Class.forName("com.mysql.cj.jdbc.Driver");
         }
         catch(ClassNotFoundException e)
     {
                  System.err.print("ClassNotFoundException: ");
                  System.err.println(e.getMessage());
     }
   }
}
```

**INSERT**

```
package com.mycompany.labexperiments;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class InsertExp {
```

```
public static void main(String[] args) {

    String jdbcUrl = "jdbc:mysql://localhost:3306/mysql";
    String username = "root";
    String password = "root";
    Connection con;

    try {
      con = DriverManager.getConnection(jdbcUrl, username, password);
      Statement stmt;
      String str, str1, str2, str3;

            str = "Insert into Alex values('Dharma', 79, 5000000)";
      str1 = "Insert into Alex values('Mani', 79, 4000000)";
      str2 = "Insert into Alex values('Alex', 69, 4500000)";
      str3 = "Insert into Alex values('Dhanush', 59, 5600000)";

      /* str =  "create table Alex " +
                        "(Name VARCHAR(32)," +
                        "Age INTEGER," +
                        "Salary FLOAT)";*/
            stmt = con.createStatement();
            stmt.executeUpdate(str);
      stmt.executeUpdate(str1);
      stmt.executeUpdate(str2);
      stmt.executeUpdate(str3);
            stmt.close();
            con.close();
  }
  catch(SQLException vc) {
    System.out.println(vc);
  }
  try
  {
             Class.forName("com.mysql.cj.jdbc.Driver");
     }
     catch(ClassNotFoundException e)
  {
            System.err.print("ClassNotFoundException: ");
```

```
                System.err.println(e.getMessage());
    }
  }
  }
```

**UPDATE**

```
package com.mycompany.labexperiments;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class UpdateExp {

   public static void main(String[] args) {

        String jdbcUrl = "jdbc:mysql://localhost:3306/mysql";
        String username = "root";
        String password = "root";
        Connection con;

        try {
           con = DriverManager.getConnection(jdbcUrl, username, password);
           Statement stmt;
           String str;

                str = "Update Alex SET Name='Ramkumar' where Age=78";

           stmt = con.createStatement();
                stmt.executeUpdate(str);
           stmt.close();
                con.close();
    }
    catch(SQLException vc) {
       System.out.println(vc);
    }
    try
    {
                 Class.forName("com.mysql.cj.jdbc.Driver");
```

```
        }
        catch(ClassNotFoundException e)
    {
                System.err.print("ClassNotFoundException: ");
                System.err.println(e.getMessage());
    }
  }
}
```

**DELETE**

```
package com.mycompany.labexperiments;

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

public class DeleteExp {

   public static void main(String[] args) {

        String jdbcUrl = "jdbc:mysql://localhost:3306/mysql";
        String username = "root";
        String password = "root";
        Connection con;

        try {
           con = DriverManager.getConnection(jdbcUrl, username, password);
           Statement stmt;
           String str;
                str = "Delete from alex where Age=67";
           stmt = con.createStatement();
                stmt.executeUpdate(str);
           stmt.close();
                con.close();
    }
```

```
    catch(SQLException vc) {
       System.out.println(vc);
    }
    try
    {
                  Class.forName("com.mysql.cj.jdbc.Driver");
        }
        catch(ClassNotFoundException e)
    {
                System.err.print("ClassNotFoundException: ");
                System.err.println(e.getMessage());
    }
  }
 }
```

**SELECT**

```
package com.mycompany.labexperiments;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;

public class SelectExp {

   public static void main(String[] args) {

        String jdbcUrl = "jdbc:mysql://localhost:3306/mysql";
        String username = "root";
        String password = "root";
        Connection con;
        ResultSet rs;
        Statement stmt;

        try {
           con = DriverManager.getConnection(jdbcUrl, username, password);
           stmt = con.createStatement();

           String str;
```

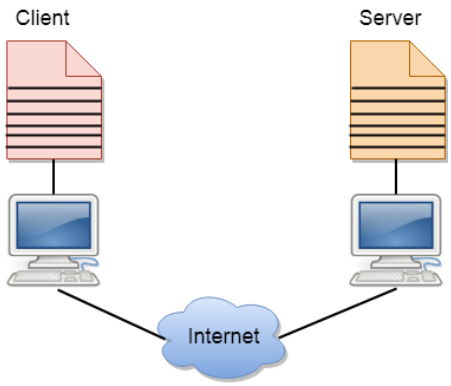
```
            str = "Select * from alex";
        //stmt.executeUpdate(str);
        rs = stmt.executeQuery(str);

        while(rs.next()){
        System.out.println("Name: " + rs.getString("Name"));
        System.out.println(", Age: " + rs.getInt("Age"));
        System.out.println(", Salary: " + rs.getLong("Salary"));
        }
        rs.close();

        stmt.close();
            con.close();
    }
    catch(SQLException vc) {
       System.out.println(vc);
    }
    try
    {
             Class.forName("com.mysql.cj.jdbc.Driver");
       }
       catch(ClassNotFoundException e)
    {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
    }
  }
}
```

**Client and Server model**

- A client and server networking model is a model in which computers such as servers provide the network services to the other computers such as clients to perform a user based tasks. This model is known as client-server networking model.
- The application programs using the client-server model should follow the given below strategies:




- An application program is known as a client program, running on the local machine that requests for a service from an application program known as a server program, running on the remote machine.
- A client program runs only when it requests for a service from the server while the server program runs all time as it does not know when its service is required.
- A server provides a service for many clients not just for a single client. Therefore, we can say that client-server follows the many-to-one relationship. Many clients can use the service of one server.
- Services are required frequently, and many users have a specific client-server application program. For example, the client-server application program allows the user to access the files, send e-mail, and so on. If the services are more customized, then we should have one generic application program that allows the user to access the services available on the remote computer.

**Client**

A client is a program that runs on the local machine requesting service from the server. A client program is a finite program means that the service started by the user and terminates when the service is completed.

**Server**

A server is a program that runs on the remote machine providing services to the clients. When the client requests for a service, then the server opens the door for the incoming requests, but it never initiates the service.

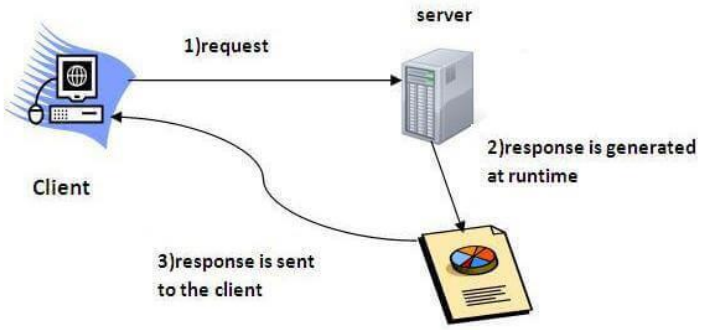
A server program is an infinite program means that when it starts, it runs infinitely unless the problem arises. The server waits for the incoming requests from the clients. When the request arrives at the server, then it responds to the request.

Advantages of Client-server networks:

- **Centralized:** Centralized back-up is possible in client-server networks, i.e., all the data is stored in a server.
- **Security:** These networks are more secure as all the shared resources are centrally administered.
- **Performance:** The use of the dedicated server increases the speed of sharing resources. This increases the performance of the overall system.
- **Scalability:** We can increase the number of clients and servers separately, i.e., the new element can be added, or we can add a new node in a network at any time.

**Servlets**

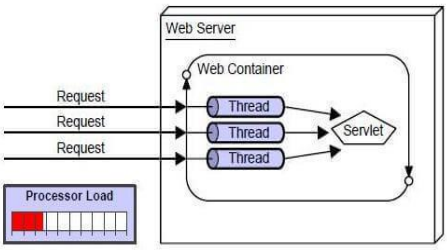
- Servlet is a technology which is used to create a web application.
- Servlet is an API that provides many interfaces and classes including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and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




**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 elements such as HTML, CSS, and JavaScript. The web components typically execute in Web Server and respond to the HTTP request.

## Advantages of Servlet



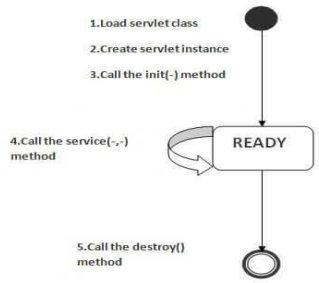

There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages of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

1. **Better performance:** because it creates a thread for each request, not process.
2. **Portability:** because it uses Java language.
3. **Robust:** JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.
4. **Secure:** because it uses java language.

## Life Cycle of a Servlet (Servlet Life Cycle)

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

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



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

**1) Servlet class is loaded**

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

**2) Servlet instance is created**

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

**3) init method is invoked**

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

**public void** init(ServletConfig config) **throws** ServletException

**4) service method is invoked**

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

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

**5) destroy method is invoked**

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

**public void** destroy()

## JSP(Java Server Pages)

**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

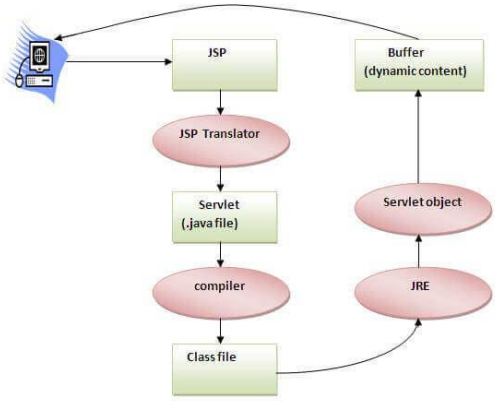
In JSP, we can use many tags such as action tags, JSTL, custom tags, etc. that reduces the code. Moreover, we can use EL, implicit objects, etc.

### 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**
**index.jsp**

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

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

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

**Running 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

**JSP Scripting elements**

The scripting elements provides the ability to insert java code inside the jsp. There are three types of scripting elements:

- scriptlet tag
- expression tag
- declaration tag

**JSP scriptlet tag**

A scriptlet tag is used to execute java source code in JSP. Syntax is as follows:

<% java source code %>

**Example of JSP scriptlet tag**

In this example, we are displaying a welcome message.

```
<html>
<body>
<% out.print("welcome to jsp"); %>
</body>
</html>
```

**JSP expression tag**

The code placed within **JSP expression tag** is written to the output stream of the response. So you need not write out.print() to write data. It is mainly used to print the values of variable or method.

Syntax of JSP expression tag

**<**%= statement %**>**

**Example of JSP expression tag**

In this example of jsp expression tag, we are simply displaying a welcome message.

```
<html>
<body>
<%= "welcome to jsp" %>
</body>
</html>
```

**Example of JSP expression tag that prints current time**

To display the current time, we have used the getTime() method of Calendar class. The getTime() is an instance method of Calendar class, so we have called it after getting the instance of Calendar class by the getInstance() method.

index.jsp

```
<html>
<body>
Current Time: <%= java.util.Calendar.getInstance().getTime() %>
</body>
</html>
```

**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:

**<**%! 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

```
<html>
<body>
<%! int data=50; %>
<%= "Value of the variable is:"+data %>
</body>
</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

```
<html>
<body>
<%!
int cube(int n){
return n*n*n*;
}
%>
<%= "Cube of 3 is:"+cube(3) %>
</body>
</html>
```

## JSP directives

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

There are three types of directives:

- page directive
- include directive
- taglib directive

## JSP page directive

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

Syntax of JSP page directive

<%@ page attribute="value" %>

**Example of import attribute**

```
<html>
<body>

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

</body>
</html>
```

## Jsp Include Directive

The include directive is used to include the contents of any resource it may be jsp file, html file or text file. The include directive includes the original content of the included resource at page translation time (the jsp page is translated only once so it will be better to include static resource).

## Advantage of Include directive

Code Reusability

Syntax of include directive

<%@ include file="resourceName" %>

**Example of include directive**

In this example, we are including the content of the header.html file. To run this example you must create an header.html file.

```
<html>
<body>

<%@ include file="header.html" %>

Today is: <%= java.util.Calendar.getInstance().getTime() %>

</body>
</html>
```

**JSP Taglib directive**

The JSP taglib directive is used to define a tag library that defines many tags. We use the TLD (Tag Library Descriptor) file to define the tags. In the custom tag section we will use this tag so it will be better to learn it in custom tag.

Syntax JSP Taglib directive

```
<%@ taglib uri="uriofthetaglibrary" prefix="prefixoftaglibrary" %>
```

---

**Example of JSP Taglib directive**

In this example, we are using our tag named currentDate. To use this tag we must specify the taglib directive so the container may get information about the tag.

```
<html>
<body>

<%@ taglib uri="http://www.javatpoint.com/tags" prefix="mytag" %>

<mytag:currentDate/>

</body>
</html>
```

**JSP Implicit Objects**

There are **9 jsp implicit objects**. These objects are created by the web container that are available to all the jsp pages.

The available implicit objects are out, request, config, session, application etc.

A list of the 9 implicit objects is given below:

| Object | Type |
|---|---|
| out | JspWriter |
| request | HttpServletRequest |
| response | HttpServletResponse |
| config | ServletConfig |
| application | ServletContext |
| session | HttpSession |
| pageContext | PageContext |
| page | Object |
| exception | Throwable |

**JSP out implicit object**

For writing any data to the buffer, JSP provides an implicit object named out. It is the object of JspWriter. In case of servlet you need to write:

PrintWriter out=response.getWriter(); But in JSP, you don't need to write this code.

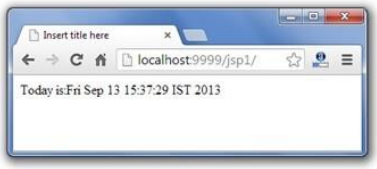
**Example of out implicit object**

In this example we are simply displaying date and time.

index.jsp
<html>

```
<body>
<% out.print("Today is:"+java.util.Calendar.getInstance().getTime()); %>
</body>
</html>
```

Output




**JavaBean**

Bean is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

**Advantages of JavaBean**

- The JavaBean properties and methods can be exposed to another application.
- It provides an easiness to reuse the software components.