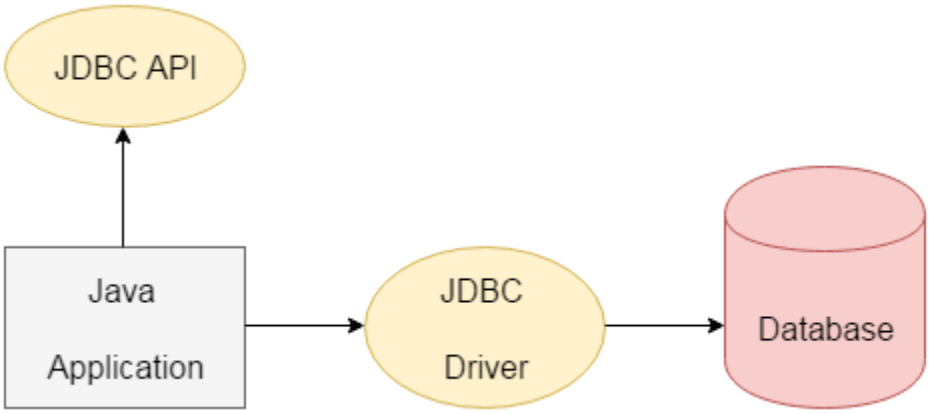
# Java JDBC

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 access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class
- Clob class
- Types class

## Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

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.

### What is API?

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

# 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 driver converts JDBC method calls into the ODBC function calls.

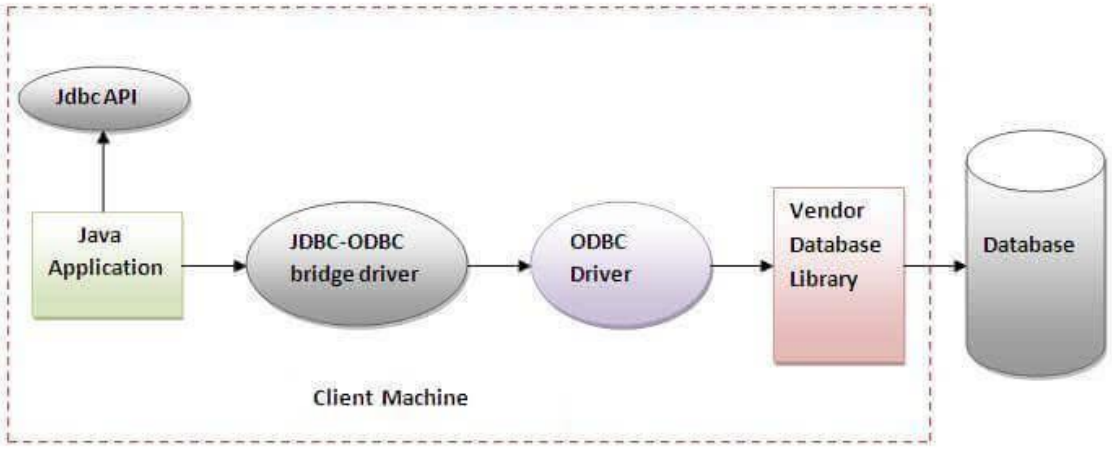



Figure- JDBC-ODBC Bridge Driver

***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.

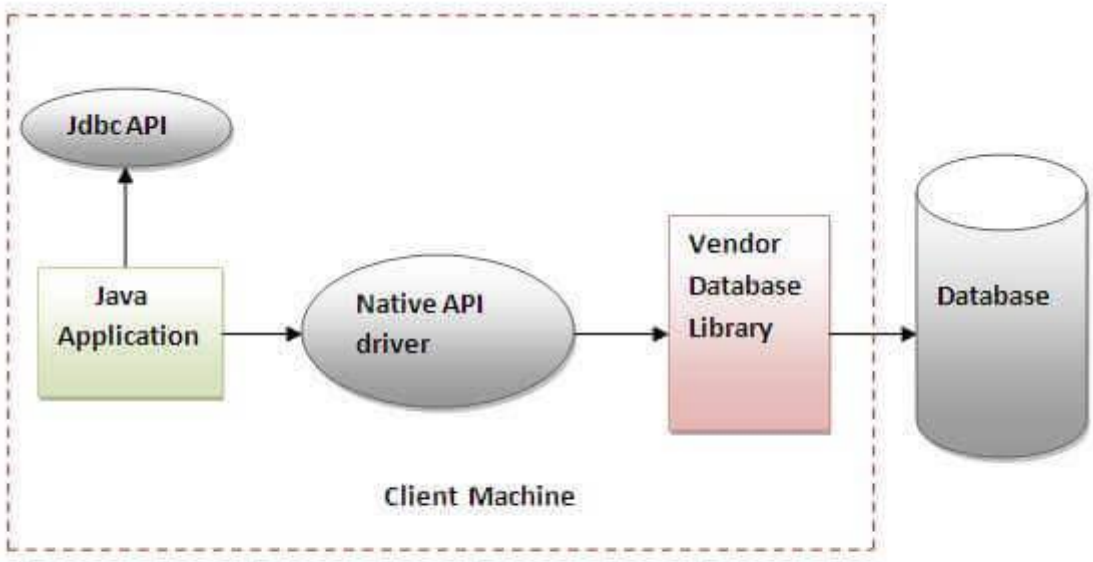



Figure- Native API Driver

- **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.

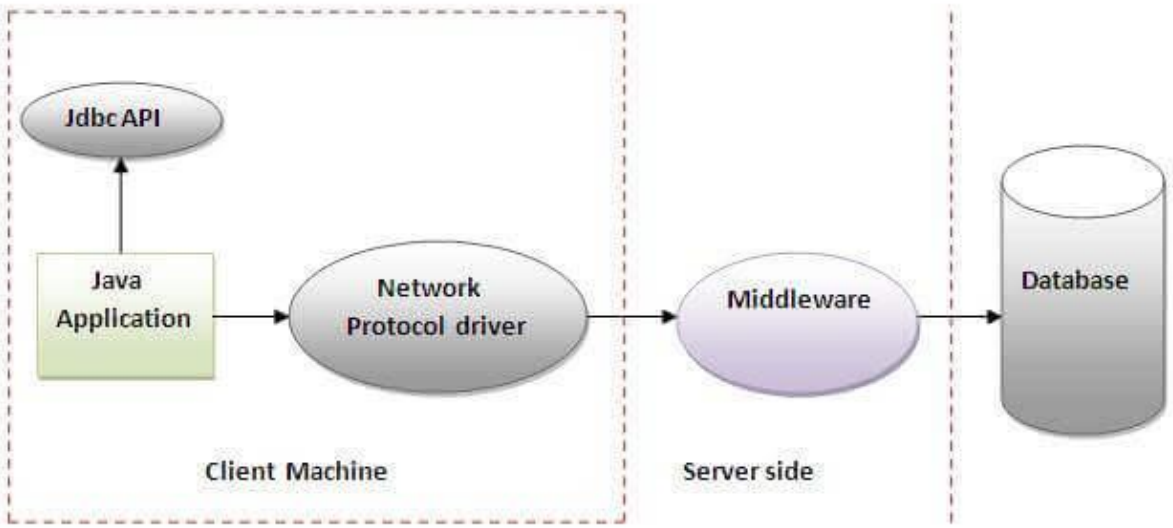



Figure- Network Protocol Driver

- **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.

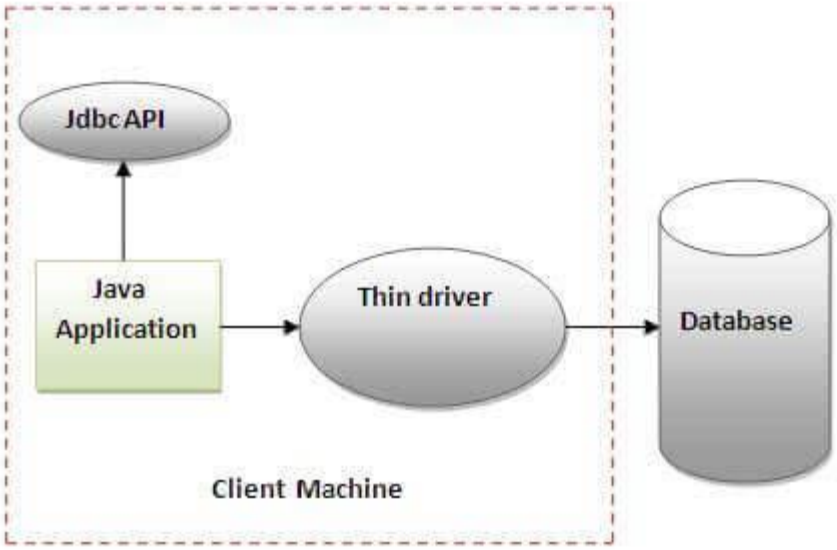



Figure- Thin Driver

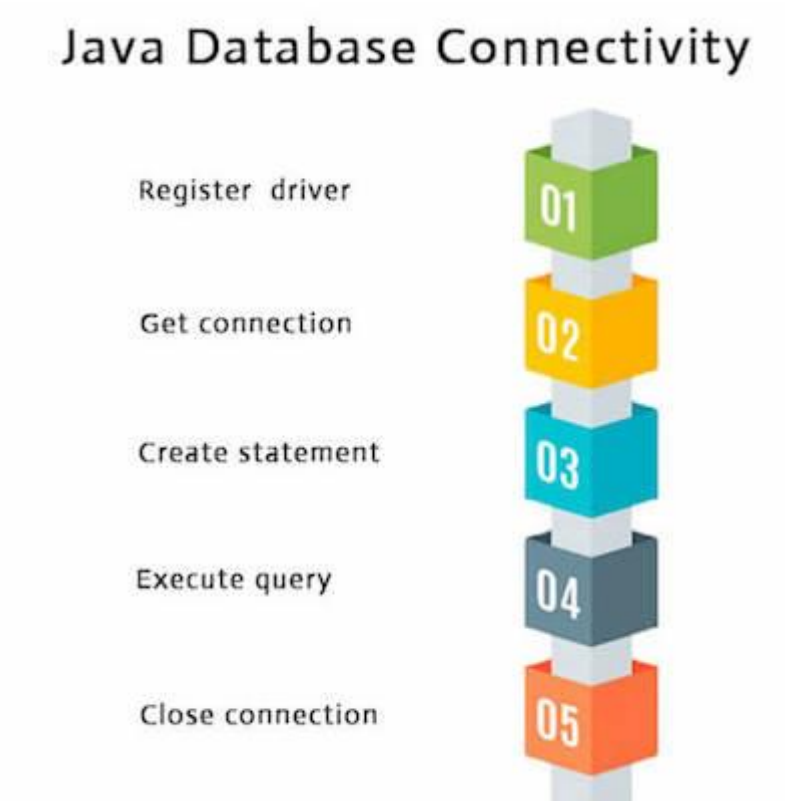
- **Advantage:**
- Better performance than all other drivers.
- No software is required at client side or server side.

- **Disadvantage:**
- Drivers depend on the Database.

# Java Database Connectivity Steps:

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 is used to register the driver class. This method is used to 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:8080:db","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**

```
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 method 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 of Connection interface is used to close the connection.

**Syntax of close() method**

```
public void close()throws SQLException
```

Example to close connection

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

# Java Database Connectivity with MySQL

To connect Java application with the MySQL database, we need to follow 5 following steps.

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and emp is the database name. We may use any database, in such case, we need to replace the emp with our database name.
3. **Username:** The default username for the mysql database is **root**.
4. **Password:** It is the password given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

```
1. create database emp;
2. use emp;
3. create table empdata(id int(10),name varchar(40),age int(3));
```

## Example to Connect Java Application with mysql database

In this example, emp is the database name, root is the username and password both.

```
1. import java.sql.*;
2. class MysqlCon{
3. public static void main(String args[]){
4. try{
5. Class.forName("com.mysql.jdbc.Driver");
6. Connection con=DriverManager.getConnection(
7. "jdbc:mysql://localhost:3306/emp","root","root");
8. //here emp is database name, root is username and password
9. Statement stmt=con.createStatement();
10.ResultSet rs=stmt.executeQuery("select * from emp");
11.while(rs.next())
12.System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
13.con.close();
14.}catch(Exception e){ System.out.println(e);}
15.}
16.}
```

To connect java application with the mysql database, **mysqlconnector.jar** file is required to be loaded.

Download mysqlconnector jar

## Two ways to load the jar file:

1. Paste the mysqlconnector.jar file in jre/lib/ext folder
2. Set classpath

Paste the mysqlconnector.jar file in JRE/lib/ext folder:

Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here.

2) Set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath

open command prompt and write:

```
C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;.;
```

How to set the permanent classpath

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;.; as C:\folder\mysql-connector-java-5.0.8-bin.jar;.;

# Statement

The Statement interface represents the static SQL statement. It helps you to create a general purpose SQL statements using Java.

### Creating a statement

You can create an object of this interface using the **createStatement()** method of the **Connection** interface.

Create a statement by invoking the **createStatement()** method as shown below.

```
Statement stmt = null;
try {
stmt = conn.createStatement( );
. . .
}
catch (SQLException e) {
. . .
}
finally {
. . .
}
```

### Executing the Statement object

Once you have created the statement object you can execute it using one of the execute methods namely, execute(), executeUpdate() and, executeQuery().

- **execute():** This method is used to execute SQL DDL statements, it returns a boolean value specifying whether the ResultSet object can be retrieved.

- **executeUpdate():** This method is used to execute statements such as insert, update, delete. It returns an integer value representing the number of rows affected.
- **executeQuery():** This method is used to execute statements that returns tabular data (example SELECT statement). It returns an object of the class ResultSet.

## Prepared Statement

The **PreparedStatement** interface extends the Statement interface. It represents a precompiled SQL statement which can be executed multiple times. This accepts parameterized SQL quires and you can pass 0 or more parameters to this query.

Initially, this statement uses place holders "**?**" instead of parameters, later on, you can pass arguments to these dynamically using the **setXXX()** methods of the **PreparedStatement** interface.

## Creating a PreparedStatement

You can create an object of the **PreparedStatement** (interface) using the **prepareStatement()** method of the Connection interface. This method accepts a query (parameterized) and returns a PreparedStatement object.

When you invoke this method the Connection object sends the given query to the database to compile and save it. If the query got compiled successfully then only it returns the object.

To compile a query, the database doesn't require any values so, you can use (zero or more) **placeholders** (Question marks "?") in the place of values in the query.

For example, if you have a table named **Employee** in the database created using the following query:

```
CREATE TABLE Employee(Name VARCHAR(255), Salary INT NOT NULL,
Location
```

```
VARCHAR(255));
```

Then, you can use a **PreparedStatement** to insert values into it as shown below.

```
//Creating a Prepared Statement
String query="INSERT INTO Employee(Name, Salary,
Location)VALUES(?, ?, ?)";
Statement pstmt = con.prepareStatement(query);
```

**Setting values to the place holders**

The **PreparedStatement** interface provides several setter methods such as setInt(), setFloat(), setArray(), setDate(), setDouble() etc.. to set values to the place holders of the prepared statement.

These methods accepts two arguments one is an integer value representing the placement index of the place holder and the other is an int or, String or, float etc... representing the value you need to insert at that particular position.

Once you have created a prepared statement object (with place holders) you can set values to the place holders of the prepared statement using the setter methods as shown below:

```
pstmt.setString(1, "Amit");
pstmt.setInt(2, 3000);
pstmt.setString(3, "Hyderabad");
```

## Executing the Prepared Statement

Once you have created the **PreparedStatement** object you can execute it using one of the **execute()** methods of the **PreparedStatement** interface namely, execute(), executeUpdate() and, executeQuery().

- **execute():** This method executes normal static SQL statements in the current prepared statement object and returns a boolean value.

- **executeQuery():** This method executes the current prepared statement and returns a **ResultSet** object.
- **executeUpdate():** This method executes SQL DML statements such as insert update or delete in the current Prepared statement. It returns an integer value representing the number of rows affected.

## CallableStatement

The **CallableStatement** interface provides methods to execute stored procedures. Since the JDBC API provides a stored procedure SQL escape syntax, you can call stored procedures of all RDBMS in a single standard way.

### Creating a CallableStatement

You can create an object of the **CallableStatement** (interface) using the **prepareCall()** method of the **Connection** interface.

This method accepts a string variable representing a query to call the stored procedure and returns a **CallableStatement** object.

A CallableStatement can have input parameters or, output parameters or, both. To pass input parameters to the procedure call you can use place holder and set values to these using the setter methods (setInt(), setString(), setFloat()) provided by the CallableStatement interface.

Suppose, you have a procedure name myProcedure in the database you can prepare a callable statement as:

```
//Preparing a CallableStatement
CallableStatement cstmt = con.prepareCall("{call myProcedure(?, ?, ?)}");
```

**Setting values to the input parameters**

You can set values to the input parameters of the procedure call using the setter methods.

These accept two arguments one is an integer value representing the placement index of the input parameter and the other is an int or,

String or, float etc… representing the value you need to pass an input parameter to the procedure.

**Note:** Instead of index you can also pass the name of the parameter in String format.

```
cstmt.setString(1, "Raghav");
cstmt.setInt(2, 3000);
cstmt.setString(3, "Hyderabad");
```

### Executing the Callable Statement

Once you have created the CallableStatement object you can execute it using one of the **execute()** method.

```
cstmt.execute();
```