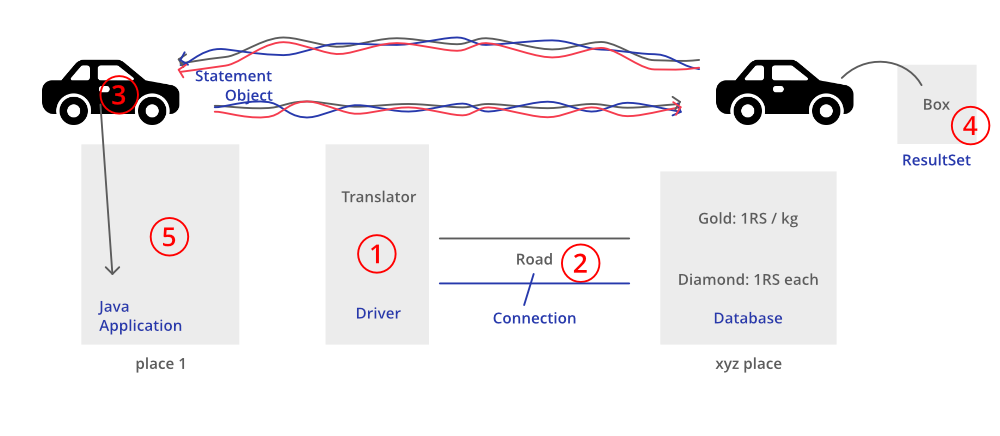
**What is JDBC?**

[JDBC](https://www.geeksforgeeks.org/introduction-to-jdbc/) is an acronym for Java Database Connectivity. It’s an advancement for ODBC ( Open Database Connectivity ). JDBC is a standard API specification developed in order to move data from frontend to the backend. This API consists of classes and interfaces written in Java. It basically acts as an interface (not the one we use in Java) or channel between your Java program and databases i.e it establishes a link between the two so that a programmer could send data from Java code and store it in the database for future use.

**Illustration:** Working of JDBC co-relating with real-time



**Why JDBC Come into Existence?**

As previously told JDBC is an advancement for ODBC, ODBC being platform-dependent had a lot of drawbacks. ODBC API was written in C, C++, Python, and Core Java and as we know above languages (except Java and some part of Python )are platform-dependent. Therefore to remove dependence, JDBC was developed by a database vendor which consisted of classes and interfaces written in Java.

**Steps For Connectivity Between Java Program and Database**

1. Import the Packages
2. Load the drivers using the *forName() method*
3. Register the drivers *using DriverManager*
4. Establish a connection*using the Connection class object*
5. Create a statement
6. Execute the query
7. Close the connections

Let us discuss these steps in brief before implementing by writing suitable code to illustrate connectivity steps for JDBC/

**Step 1:** Import the Packages

**Step 2:**Loading the drivers

In order to begin with, you first need to load the driver or register it before using it in the program. Registration is to be done once in your program. You can register a driver in one of two ways mentioned below as follows:

**2-A**Class.forName()

Here we load the driver’s class file into memory at the runtime. No need of using new or create objects. The following example uses Class.forName() to load the Oracle driver as shown below as follows:

Class.forName(“oracle.jdbc.driver.OracleDriver”);

**2-B**DriverManager.registerDriver()

DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time. The following example uses DriverManager.registerDriver()to register the Oracle driver as shown below:

DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())

**Step 3:**Establish a connection*using*the *Connection class object*

After loading the driver, establish connections as shown below as follows:

Connection con = DriverManager.getConnection(url,user,password)

* **user: U**sername from which your SQL command prompt can be accessed.
* **password:**password from which the SQL command prompt can be accessed.
* **con:** It is a reference to the Connection interface.
* **Url**: Uniform Resource Locator which is created as shown below:

String url = “ jdbc:oracle:thin:@localhost:1521:xe”

Where oracle is the database used, thin is the driver used, @localhost is the IP Address where a database is stored, 1521 is the port number and xe is the service provider. All 3 parameters above are of String type and are to be declared by the programmer before calling the function. Use of this can be referred to form the final code.

**Step 4:**Create a statement

Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database.   
Use of JDBC Statement is as follows:

Statement st = con.createStatement(“select \* from databasename.emp;”);

***Note:****Here, con is a reference to Connection interface used in previous step .*

**Step 5:**Execute the query

Now comes the most important part i.e executing the query. The query here is an SQL Query. Now we know we can have multiple types of queries. Some of them are as follows:

* The query for updating/inserting a table in a database.
* The query for retrieving data.

The executeQuery() method of the **Statement interface** is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.   
The executeUpdate(sql query) method of the Statement interface is used to execute queries of updating/inserting.

**Pseudo Code:**

int m = st.executeUpdate(sql);

if (m==1)

System.out.println("inserted successfully : "+sql);

else

System.out.println("insertion failed");

Here sql is SQL query of the type String

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. This is now discouraged because of thin 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. |



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.



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. |



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

|  |
| --- |
| 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

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. 1) **public** **static** Connection getConnection(String url)**throws** SQLException
2. 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");
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

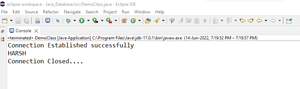
Example to close connection

1. con.close();

* Java

|  |
| --- |
| // This code is for establishing connection with MySQL  // database and retrieving data  // from db Java Database connectivity    /\*   \*1. import --->java.sql   \*2. load and register the driver ---> com.jdbc.   \*3. create connection   \*4. create a statement   \*5. execute the query   \*6. process the results   \*7. close   \*/    **import** java.io.\*;  **import** java.sql.\*;    **class** GFG {  **public** **static** **void** main(String[] args) **throws** Exception      {          String url              = "jdbc:<mysql://localhost:3306/table_name>"; // table details          String username = "rootgfg"; // MySQL credentials          String password = "gfg123";          String query              = "select \*from students"; // query to be run          Class.forName(              "com.mysql.cj.jdbc.Driver"); // Driver name          Connection con = DriverManager.getConnection(              url, username, password);          System.out.println(              "Connection Established successfully");          Statement st = con.createStatement();          ResultSet rs              = st.executeQuery(query); // Execute query          rs.next();          String name              = rs.getString("name"); // Retrieve name from db            System.out.println(name); // Print result on console          st.close(); // close statement          con.close(); // close connection          System.out.println("Connection Closed....");      }  } |

**Output:**



*name of the student is retrieved from database*

**Step 6: Closing the connections**

So finally we have sent the data to the specified location and now we are on the verge of completing our task. By closing the connection, objects of Statement and ResultSet will be closed automatically. The close() method of the Connection interface is used to close the connection. It is shown below as follows:

con.close();

**Example:**

* Java

|  |
| --- |
| // Java Program to Establish Connection in JDBC    // Importing database  **import** java.sql.\*;  // Importing required classes  **import** java.util.\*;    // Main class  **class** Main {        // Main driver method  **public** **static** **void** main(String a[])      {            // Creating the connection using Oracle DB          // Note: url syntax is standard, so do grasp          String url = "jdbc:oracle:thin:@localhost:1521:xe";            // Username and password to access DB          // Custom initialization          String user = "system";          String pass = "12345";            // Entering the data          Scanner k = **new** Scanner(System.in);            System.out.println("enter name");          String name = k.next();            System.out.println("enter roll no");  **int** roll = k.nextInt();            System.out.println("enter class");          String cls = k.next();            // Inserting data using SQL query          String sql = "insert into student1 values('" + name                       + "'," + roll + ",'" + cls + "')";            // Connection class object          Connection con = **null**;            // Try block to check for exceptions  **try** {                // Registering drivers              DriverManager.registerDriver(  **new** oracle.jdbc.OracleDriver());                // Reference to connection interface              con = DriverManager.getConnection(url, user,                                                pass);                // Creating a statement              Statement st = con.createStatement();                // Executing query  **int** m = st.executeUpdate(sql);  **if** (m == 1)                  System.out.println(                      "inserted successfully : " + sql);  **else**                  System.out.println("insertion failed");                // Closing the connections              con.close();          }            // Catch block to handle exceptions  **catch** (Exception ex) {              // Display message when exceptions occurs              System.err.println(ex);          }      }  } |

**Output:**

