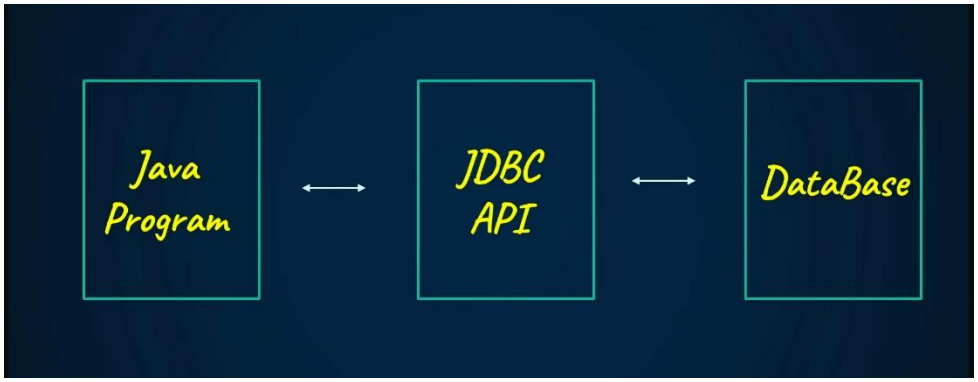
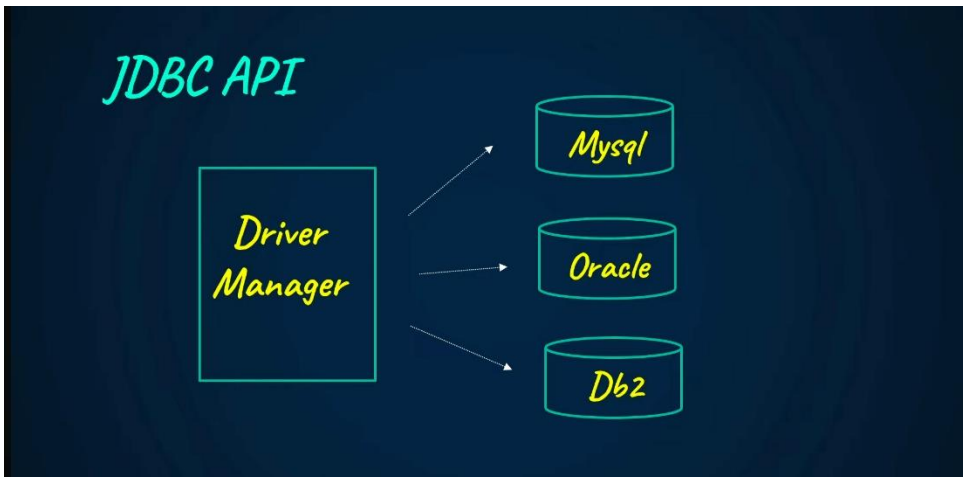
# JDBC Connectivity Tutorial

JDBC acts as an interface between the Core Java and the Database.




An interface is the one between two or more things that enables them to communicate with each other.

JDBC Java Database Connectivity, which is an API (Application Programming Interface) that is a bridge that is a software between the other two software.




Driver Manager has many drivers for different databases.

In Java, we give a connection string to JDBC for connecting them by giving their password and username.

The driver manager only checks which database we need to connect then it takes the connection and sets it up.

If, for example, we are telling MySQL connection then it uses the MySQL database driver. The MySQL driver only contains all the logic for converting the Java code into SQL code.

**Types of drivers:**


1) It is written in the C language created by Microsoft. In this, the things written in Java are converted to C for the ODBC, then make a database connection.
2) Database providers made Api that are native API in both 1 and 2. The client needs to install the application on their system, like installing a native API or ODBC, then you should use it.
3) They used middleware using network sockets where the code is sent there, then the codes are transferred to the database, so no need to install.
4) The direct conversion is the one that has full Java in it.

**Import java.sql.*;**

It is the one that has all the classes and interfaces in it.

**Connections in JDBC:**

Connection is the interface to create a connection.

So, we cannot create an object directly from it because it is like a **contract** or **blueprint** that only **declares methods**, but **does not provide their implementation**. So, when you are doing so, you need some other class for that, so that we take DriverManager over here and take the method in that called as getConnection().

**getConnection():**

can have 3 parameters like url, username, and password.

**url=”jdbc:mysql://localhost:3306/jdbcdemo”;**

The statement interface is created through which all the queries are executed.

**Createstatement** is created in order to create the query and execute it.

To execute the query, we use **executeQuery** which gives us a result set after executing. So, we create a **result set** object and receive the query.

Execute query is used when the records are going to be read.

**executeUpdate** is used when records are updated in the database, and it returns an integer that tells how many records are affected.

**PreparedStatement** is used for inserting many variables in the query rather than inserting one by one. It is a class that is used when we insert the values from the console, then we need to add **???** in the query, then use a prepared statement.

Tell which question mark belongs to which statement by using the datatype and column number we are going to insert.

While using this, we do not need to put a query in the brackets because it is already prepared.

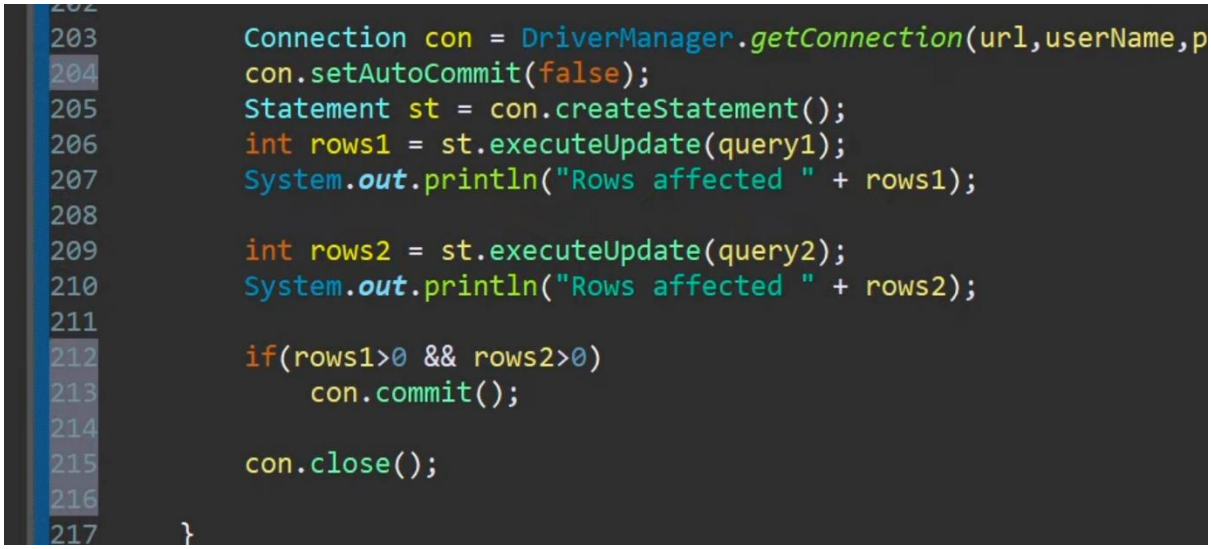
**Types of statement:**


1) **Createstatement** is used when we create the query for the select purpose
2) **Preparedstatement** is used when we insert the data into it
3) **CallableStatement** is used with a stored procedure. In this, we have **input parameters** and **output parameters**, and when the output parameter is there, then we need to give executeUpdate

**Commit vs auto commit:**

Auto commit updates the values directly in the database even though we did not tell it.

By default, the auto commit is true, so if the 1st row does not have any issues, then it is committed automatically.



```
        Connection con = DriverManager.getConnection(url,userName,p
        con.setAutoCommit(false);
        Statement st = con.createStatement();
        int rows1 = st.executeUpdate(query1);
        System.out.println("Rows affected " + rows1);

        int rows2 = st.executeUpdate(query2);
        System.out.println("Rows affected " + rows2);

        if(rows1>0 && rows2>0)
            con.commit();

        con.close();

    }
```

**Batch processing:**

Combine the several processes and establish them in one run.

To add that **addBatch** we can use, and to execute it, we can use **executeBatch**.

In this, as we use a loop, we can use **rollback** so that the previous ones are not committed until we tell it.