JDBC (Java Database Connectivity)

- It is a standard API specification developed in order to move data from frontend i.e your Java Program to backend end i.e the database.
- This API consists of classes and interfaces written in Java.
- Include the following java package in the java program to use the above classes and interfaces.
  import java.sql.*; //It contains the interface definitions for the functionality provided by jdbc

Basic Steps for connectivity between Java program and database

- Loading the **Driver**
- Establish a **Connection**
- Create JDBC **Statements**
- Execute **SQL** Statements
- **Close** connections

- **Loading the Driver**
  - Each database product that supports JDBC provides a JDBC driver, which must be dynamically loaded in order to access the database from Java.
  - This is done by invoking the Class.forname() method as follows:

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

  - The oracle driver is available in a .jar file at vendor web sites and should be placed within the classpath so that the Java compiler can access it.
- **Establish the connections**
  - After loading the driver the connection is opened using the getConnection() method of the Driver-Manager class(within java.sql) as follows:

  Connection con=DriverManager.*getConnection*(conurl,"userid","password");

  - Connection interface represents an established database connection from which we can create statements to execute queries and retrieve results, get metadata about the database, close connection, etc.
  - The getConnection()method takes three parameters:
    - url where the server runs,

  - database user identifier
  - password

- First parameter is an URL or machine name, where the server runs. Here it is

  String conurl="jdbc:oracle:thin:@172.17.144.110:1521:ora11g";

  oracle is the database used.

  thin is the driver used.

  :@172.17.144.110 is the IP Address where database is stored

  1521 is the port number the database system uses for communication

  ora11g is the specific database on the server to be used.

All 3 parameters of getConnection() are of String type and are to be declared by programmer before calling the function.

- **Create JDBC Statements**

  - Once a connection is established, the program can use it to send SQL statements to the database system for execution.
  - The JDBC Statement, PreparedStatement, and CallableStatement interfaces define the methods and properties that enable us to send SQL or PL/SQL commands and receive data from the database.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

  - Use of JDBC Statement is as follows:

    Statement stmt=con.createStatement();

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

- A Statement object is not the SQL statement itself, but rather an object that allows the Java program to invoke methods that ship an SQL statement given as an argument for execution by the database system.
- Once a Statement object is created, it can be used to execute an SQL statement with one of its three execute methods.

boolean execute (String SQL): Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

int executeUpdate (String SQL): Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.

ResultSet executeQuery (String SQL): Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

- **Execute SQL Statements**

  - Query for retrieving data .
  - Query for inserting, deleting, updating table in a database

Display all the records of customer table.

```
String sqlstr="select * from CUSTOMER";
ResultSet rs=stmt.executeQuery(sqlstr);
```

- The stmt.executeQuery method retrieves the set of tuples in the result into a ResultSet object **rs** and fetches them one tuple at a time.

Fetching one tuple at a time using rs object

```
while(rs.next())
        {
        System.out.print(rs.getString(1));
        System.out.print(rs.getString(2));
        System.out.print(rs.getString(3));
        System.out.println(rs.getString(4));
        }
```

- The next method on the result set tests whether or not there remains atleast one unfetched tuple in the result set and if so, fetches it. The return value of next method is a Boolean indicating whether it fetched a tuple.
- Attributes from the fetched tuple are retrieved using various methods whose name begins with get followed by the datatype. (Example: getString(), getInt(), getFloat(), getLong()).
- The argument to the various get methods can either be an attribute name specified as a String or an integer indicating the position of the desired attributes in a tuple.

- **Close the connections**

  - The statement and connection are both closed at the end of the java program using close method as follows:

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

Steps to add new record to a table:

- Find the number of fields of the table
  Customer(cust_no, name, phone_no, city)

- Write statements to take user input for those variables

```
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter cust_no: \t");
       String cust_no=br.readLine();
       System.out.println("Enter phone_no: \t");
       long phone_no=Long.parseLong(br.readLine());
```

- Create a statement

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

- Write a string for insert statement

```
String insertstmt="insert into customer values(' " +cust_no+" ',' "+name+" ',"
+phone_no+",' "+city+" ' )";
```

- Execute the insert statement using executeUpdate method returning n number of rows

```
int n=stmt.executeUpdate(insertstmt);
System.out.println(n+" row inserted");
```

Steps to delete a record from a table:

- Write statements to take user input for one field

```
System.out.println("Enter cust_no: \t");
String cust_no=br.readLine();
```

- Create a statement

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

- Write a string for delete statement

```
String deletestmt = "delete from customer where cust_no=' "+cust_no+" ' ";
```

- Execute the delete statement using executeUpdate method returning n number of rows

```
n=stmt.executeUpdate(deletestmt);
System.out.println(n+" row deleted");
```

Steps to update a record in a table:

- Write statements to take user input for one field

```
System.out.println("Enter cust_no: \t");
String cust_no=br.readLine();
```

- Write statement to take user input for the updated field

```
System.out.println("Enter the phone number to be updated");
long pno=Long.parseLong(br.readLine());
```

- Create a statement

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

- Write a string for updtae statement

```
String updatestmt = "update customer set phone_no="+pno+" where
cust_no=' "+cust_no+" ' ";
```

- Execute the update statement using executeUpdate method returning n number of rows

```
n=stmt.executeUpdate(updatestmt);
System.out.println(n+" row updated");
```