Tutorial on JDBCODBC

Establishing JDBC Connection in JAVA

We are going to establish a connection between front end i.e. our Java Program and back end i.e the database. But before that we should learn what precisely a JDBC is and why we need it?

What?

JDBC is an acronym for Java Database Connectivity. It's an advancement for ODBC (Open Database Connectivity). JDBC is a standard API specification developed to move data from frontend to 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 our 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.

Why?

As JDBC is an advancement for ODBC, ODBC being platform dependent had a lot of drawbacks. ODBC API was written in C, C++, Python, 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 database vendors which consists of classes and interfaces written in Java to fulfill the requirements and objectives.

STEPS FOR CONNECTIVITY BETWEEN JAVA PROGRAM AND DATABASE

1. Loading the Driver

To begin with, we first need to load the driver or register it before using it in the program. Registration is to be done once in your program. We can register a driver in one of two ways mentioned below:

- **Class.forName():** Here we load the driver's class file into memory at the runtime. No need of using new or creation of object. The following example uses Class.forName() to load the Oracle driver –

```
Class.forName("com.mysql.jdbc.Driver");
```

2. Create the Connections

After loading the driver, establish the connection between java program and database using:

```
Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/bennett", "root","123456");
```

Where:

user – username from which your sql command prompt can be accessed.
password – password from which your sql command prompt can be accessed.

con: is a reference to Connection interface.

3. Create a statement

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

Statement st = con.createStatement();

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

More on Statement:

Once a connection is obtained we can interact with the database. The JDBC *Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable us to send SQL or PL/SQL commands and receive data from our database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	**Use of Interface**
Statement	Use the 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 the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters

A. The Statement Objects

Creating Statement Object

Before we can use a Statement object to execute a SQL statement, we need to create one using the Connection object's createStatement( ) method, as in the following example −

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
// creation of statement object
   . . .}
catch (SQLException e) {
   . . .}
finally {
stmt.close(); }     // closing Statement object
```

Once you've created a Statement object, you can then use it 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 several 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.

Closing Statement Object

Just as we close a Connection object to save database resources, for the same reason we should also close the Statement object.

A simple call to the close() method will do the job. If we close the Connection object first, it will close the Statement object as well. However, we should always explicitly close the Statement object to ensure proper cleanup.

B. The PreparedStatement Objects

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
```

```
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL); // creating PreparedStatement object
   . . }
catch (SQLException e) {
   . . .}
finally {
   pstmt.close();    // closing object
}
```

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

C. The CallableStatement Objects

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object

The following code snippet shows how to employ the Connection.prepareCall() method to instantiate a CallableStatement object

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL); // creating CallableStatement object
   . . . }
catch (SQLException e) {
   . . .
}
finally {
   cstmt.close(); // closing object
}
```

4. Execute the query

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

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

The executeQuery() method of 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 Statement interface is used to execute queries of updating/inserting.

Example:

```
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.

5.Close the connections

So finally, we have sent the data to the specified location and now we are at the completion of our task. By closing connection, objects of Statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Example :

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

Example 1: Implementation of JDBC Connection

```
Import java.sql.*;
Import java.util.*;
class Main
{
        public static void main(String a[])
        {
                //Creating the connection
                String url = "dbc:mysql://localhost:3306/bennett", "root", "123456"";
                String user = "root";
                String pass = "123456";

                //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 con=null;
                try
                {
                        DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

                        //Reference to connection interface
                        con = DriverManager.getConnection(url,user,pass);
                        Statement st = con.createStatement();
                        int m = st.executeUpdate(sql);
                        if (m == 1)

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

        System.out.println("insertion failed");
                        con.close();
                }
                catch(Exception ex)
                {

        System.err.println(ex);
                }
        }
}
```