JDBC (Java Database Connectivity)

1. Introduction to JDBC

JDBC or Java Database Connectivity is a Java API to connect and execute the query with the database. It is a specification from Sun microsystems that provides a standard abstraction (API or Protocol) for java applications to communicate with various databases. It provides the language with java database connectivity standards. It is used to write programs required to access databases.

Components of JDBC

There are generally four main components of JDBC through which it can interact with a database. They are as mentioned below:

1. JDBC API: It provides various methods and interfaces for easy communication with the database. It provides two packages as follows, which contain the java SE and Java EE platforms to exhibit WORA(write once run everywhere) capabilities.

java.sql.*; and javax.sql.*

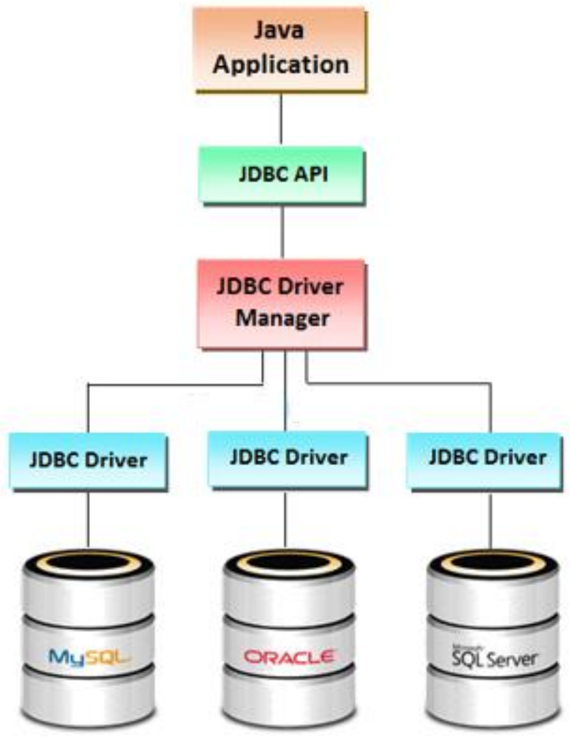
2. It also provides a standard to connect a database to a client application.

3. JDBC Driver manager: It loads a database-specific driver in an application to establish a connection with a database. It is used to make a database-specific call to the database to process the user request.

4. JDBC Test suite: It is used to test the operation(such as insertion, deletion, updation) being performed by JDBC Drivers.

5. JDBC-ODBC Bridge Drivers: It connects database drivers to the database. This bridge translates the JDBC method call to the ODBC function call. It makes use of the sun.jdbc.odbc package which includes a native library to access ODBC characteristics.

2. Architecture of JDBC



Application: It is a java program that communicates with a data source.

The JDBC API: The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important classes and interfaces defined in JDBC API are as follows:

DriverManager: It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.

JDBC drivers: To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

3. JDBC Drivers

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver

a. Type-1 driver

Type-1 driver or 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. Type-1 driver is also called Universal driver because it can be used to connect to any of the databases.

- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.
- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver isn't written in java, that's why it isn't a portable driver.
- This driver software is built-in with JDK so no need to install separately.
- It is a database independent driver.

b. Type-2 driver

The Native API driver uses the client -side libraries of the database. This driver converts JDBC method calls into native calls of the database API. In order to interact with different database, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 driver.

- Driver needs to be installed separately in individual client machines
- The Vendor client library needs to be installed on client machine.
- Type-2 driver isn't written in java, that's why it isn't a portable driver
- It is a database dependent driver.

c. Type-3 driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation.

- Type-3 drivers are fully written in Java, hence they are portable drivers.
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
- Network support is required on client machine.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.
- Switch facility to switch over from one database to another database.

d. Type-4 driver

Type-4 driver is also called native protocol driver. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver.

- Does not require any native library and Middleware server, so no client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.

4. Statement

The statement interface is used to create SQL basic statements in Java it provides methods to execute queries with the database.

1. **Create a Statement:** From the connection interface, you can create the object for this interface. It is generally used for general–purpose access to databases and is useful while using static SQL statements at runtime.

Syntax:

Statement statement =
connection.createStatement();

Implementation: Once the Statement object is created, there are three ways to execute it.

1. boolean execute(String SQL): If the ResultSet object is retrieved, then it returns true else false is returned. Is used to execute SQL DDL statements or for dynamic SQL.

2. int executeUpdate(String SQL): Returns number of rows that are affected by the execution of the statement, used when you need a number for INSERT, DELETE or UPDATE statements.

3. ResultSet executeQuery(String SQL): Returns a ResultSet object. Used similarly as SELECT is used in SQL.

5. PreparedStatement

Prepared Statement represents a recompiled SQL statement, that can be executed many times. This accepts parameterized SQL queries. In this, "?" is used instead of the parameter, one can pass the parameter dynamically by using the methods of PREPARED STATEMENT at run time.

Considering in the people database if there is a need to INSERT some values, SQL statements such as these are used:

INSERT INTO people VALUES ("Ayan",25);
INSERT INTO people VALUES("Kriya",32);

To do the same in Java, one may use Prepared Statements and set the values in the ? holders, setXXX() of a prepared statement is used as shown:

String query = "INSERT INTO people(name, age)VALUES(?, ?)";
Statement pstmt = con.prepareStatement(query);
pstmt.setString(1,"Ayan");
ptstmt.setInt(2,25);
// where pstmt is an object name

Implementation: Once the PreparedStatement object is created, there are three ways to execute it:

1. execute(): This returns a boolean value and executes a static SQL statement that is present in the prepared statement object.

2. ***executeQuery():*** Returns a ResultSet from the current prepared statement.

3. ***executeUpdate():*** Returns the number of rows affected by the DML statements such as INSERT, DELETE, and more that is present in the current Prepared Statement.

6. ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

ResultSet Interface is present in the java.sql package. It is used to store the data which are returned from the database table after the execution of the SQL statements in the Java Program.

The object of ResultSet maintains cursor point at the result data. In default, the cursor positions before the first row of the result data.

The next() method is used to move the cursor to the next position in a forward direction. It will return FALSE if there are no more records. It retrieves data by calling the executeQuery() method using any of the statement objects.

Statement Interface

```
Statement statemnt1 = conn.createStatement();
ResultSet rs1 = statemnt1.executeQuery(“Select * from EMPLOYEE_DETAILS”);
```

ResultSet Types

In default, we can iterate the data/values in ResultSet which have returned as an output of the executed SQL statement in the forward direction. We can iterate the values in other directions using Scrollable ResultSet. We can specify the type and concurrency of ResultSet while creating Statement, PreparedStatement, and CallableStatement objects.

There are 3 types in ResultSet. They are:

a. **TYPE_FORWARD_ONLY:** It is the default option, where the cursor moves from start to end i.e. in the forward direction.
b. **TYPE_SCROLL_INSENSITIVE:** In this type, it will make the cursor to move in both forward and backward directions. If we make any changes in the data while iterating the stored data it won’t update in the dataset if anyone changes the data in DB. Because the dataset has the data from the time the SQL query returns the Data.
c. **TYPE_SCROLL_SENSITIVE:** It is similar to TYPE_SCROLL_INSENSITIVE, the difference is if anyone updates the data after the SQL Query has returned the data, while iterating it will reflect the changes to the dataset.

7. ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc.

ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

```
PreparedStatement ps=con.prepareStatement("select * from emp");
ResultSet rs=ps.executeQuery();
ResultSetMetaData rsmd=rs.getMetaData();
```

```java
System.out.println("Total columns: "+rsmd.getColumnCount());
System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));
```

Inserting Records to database server

```
import java.sql.*;
class InsertPrepared{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
stmt.setInt(1,101);//1 specifies the first parameter in the query
stmt.setString(2,"Ratan");
int i=stmt.executeUpdate();
System.out.println(i+" records inserted");
con.close();
}
catch(Exception e)
{
 System.out.println(e);
} } }
```

PreparedStatement interface that updates the record

```
PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
stmt.setString(1,"Sonoo");//1 specifies the first parameter in the query i.e. name
stmt.setInt(2,101);
int i=stmt.executeUpdate();
System.out.println(i+" records updated");
```

PreparedStatement interface that deletes the record

```
PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");
stmt.setInt(1,101);
int i=stmt.executeUpdate();
System.out.println(i+" records deleted");
```

PreparedStatement interface that retrieve the records of a table

```
PreparedStatement stmt=con.prepareStatement("select * from emp");
ResultSet rs=stmt.executeQuery();
while(rs.next())
{
        System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```