Java Database Connectivity (JDBC)

JDBC- A Technical Introduction

JDBC is a JAVA Database API, that lets a programmer access virtually any tabular data source i.e. database management system from a Java application.

JDBC not only provides connectivity to a wide range of SQL databases, JDBC allows you to access other tabular data sources such as SPREAD SHEETS or FLATFILES.

JDBC is often thought as an acronym for Java Database Connectivity, but the trademarked API name is JDBC.

JDBC defines a low-level API designed to support basic SQL functions independent of any SQL implementation.

JDBC is based on X/Open Call Level Interface, an international standard for programming access to SQL databases, which is also basic for Microsoft ODBC interface.

JDBC 2.0 API includes two packages.

java.sql -> Jdbc 2.0 core API

javax.sql -> Jdbc Standard Extension.

Both the packages together contain the classes & interfaces required to create a database application using Java.

- javax.sql is an advanced package, also known as Jdbc 3.0 API, which have more enhanced feature than that of Jdbc 2.0 API, such as CONNECTION POOLING & ROWSETS.
- Main Strength of Jdbc is that, it is designed to work exactly in a same manner with any relational database.
- In other works, one program will work for any database.
- Jdbc provides a uniform interface on top of variety of different database-connectivity module.

- **Pre-Requisites for JDBC**
  - Creating DSN.
  - Basic SQL Commands.

Creating DSN (For TYPE 1 DRIVER)

Steps for creating DSN

Control Panel → System & Security → Administrative Tools →Data Sources (ODBC)→ System DSN→ Add→ SQL Server → Fill Information →Map the Data Source name With a specified database→Finish.

Basic SQL Commands. (With context to MySQL Database)

- **Creating a Database**

  Create Database Database_Name

  Create Database Student;

- **Creating a Table**

  Create Table Table_Name

  (column_name datatype)

  Create Table Student_TYIT

  (S_ID int,

  S_Name varchar(20),

  S_Class varchar(30)

  );

- **Inserting Values Into Table**

  Insert Into Table_Name

  Values(values accordingly)

```
Insert Into Student_TYIT
Values(1,'TOM','TYIT');
```

- **Deleting from a Table**

Delete

From Table_Name

***Where condition**

***-**Optional

```
Delete
From Student_TYIT
Where S_ID=1;
```

- **Selecting Values from a Table**

Select */column_name

From Table_Name

Where condition.

```
Select *
From Student_TYIT

Select S_Name
From Student_TYIT
Where S_ID=1;
```

- **Updating Values IN a Table**

Update Table_Name

Set column_Name=new_value

Where condition

Update Student_TYIT

Set S_Name='Alex'

Where S_ID=1;

- **Dropping a Table**

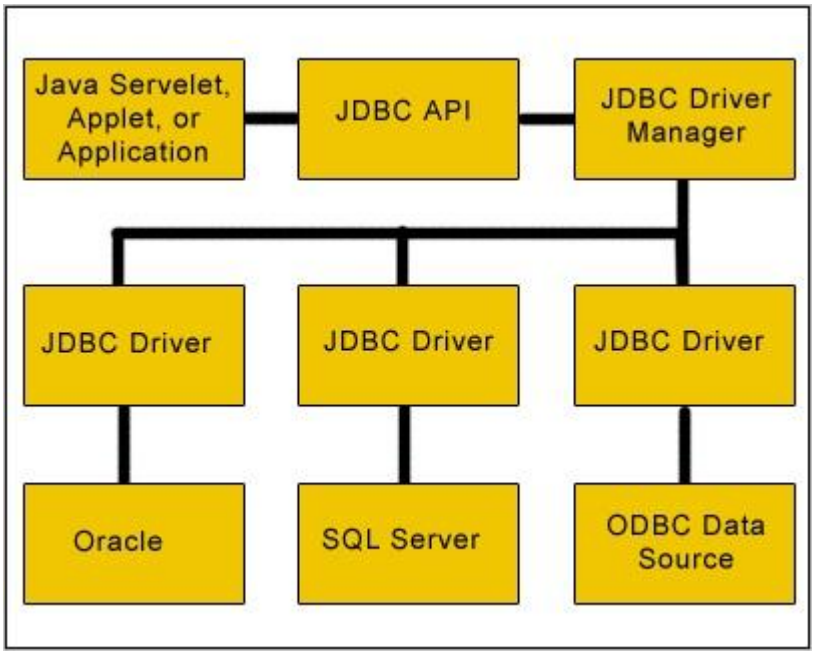
Drop table table_Name

Drop table Student_TYIT;

Note: DSN creation is according to Microsoft SQL SERVER

JDBC-Architecture

- ✓ JDBC is an API (Application Programming Interface) developed by Sun Microsystem.
- ✓ It provides a definition of a uniform interface for accessing various relational database.
- ✓ Package java.sql i.e. JDBC 2.0 is the core part of Java platform.
- ✓ First function of JDBC API is to provide a way for the developer to use SQL statements & process the result in such a manner that result is consistent & the connectivity remains database-independent.
- ✓ JDBC provides rich, object oriented access to databases by defining classes & interfaces that represents objects such as
  - ➢ Database Connections
  - ➢ Database Drivers
  - ➢ Driver Manger
  - ➢ SQL Statements
  - ➢ Prepared Statement
  - ➢ Callable Statement
  - ➢ ResultSet
  - ➢ ResultSetMetaData
  - ➢ DatabaseMetaData



- ✓ JDBC API uses DriverManager & Database specific Drivers to provide transparent connectivity to a variety of relational database.
- ✓ JDBC DriverManager is capable of supporting multiple concurrent drivers connected to different databases.
- ✓ JDBC Driver translates standard JDBC calls into NETWORK SPECIFIC or DATABASE PROTOCOL which facilitates the communication with database.
- ✓ This translation process helps JDBC application to be database independent.

JDBC-Driver Model

- ✓ JDBC API supports two types of models for accesing a database.
  - Two-Tier Model
  - Three-Tier Model
- ✓ Jdbc can either directly be used as a part of user application or as a part of a middle-tier server application.

- **Two-Tier Model**

- ➢ In Two-Tier model, a Java application interacts directly with the database.

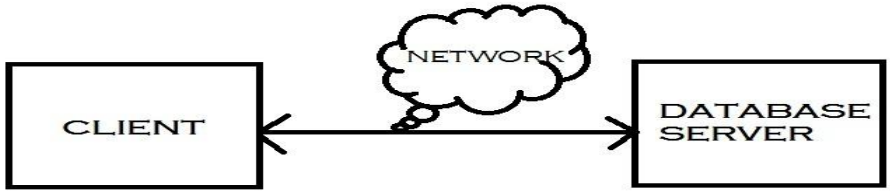


FIG: TWO TIER MODEL

- ➢ Functionality is divided into two layers.

  Application Layer: Includes Jdbc driver, Business logic & user interface.
  Database Layer: Includes RDBMS.

- ➢ Interface to the database is handelled by JDBC driver appropriate to the particular DBMS that is being accessed.
- ➢ JDBC driver passes SQL statements to the database & returns the result of those statements to the application.
- ➢ Client-server configuration is a special case of Two-Tier model, where database is located on another machine called as server.
- ➢ Application runs on Client machine which is connected to a server over a network.

- **Three-Tier Model**

- In Three-Tier model, a Java application interacts with middle tier. i.e. application layer, and that layer in turn interacts with database.

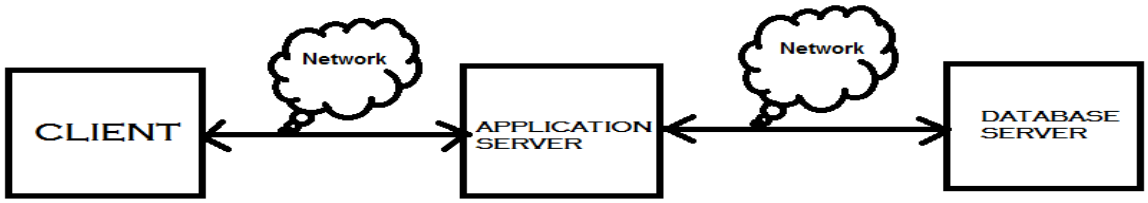


FIG: THREE TIER MODEL

- Java Application/ Client sends the SQL statements to the application server & application server sends the request to the databse.
- Database processes the SQL statements & sends the result back to application server, which sends it to client.
- Main components of three-tier architecture are

  Client-Tier: Typically a thin presentation layer, that may be implemented using Web-Browser.

  Application Tier/Middle Tier: Handles business logic or application logic. Can be implemented using web-server such as Tomcat or Web-Sphere. JDBC Drivers resides inside them.

  Database Tier : Includes Database Server

- Three-Tier model is most common in web-application where Client-Tier is frequently implemented in a browser on client machine, middle tier is implemented in a web-server & DBMS runs on dedicated database server.

JDBC-DRIVERS

JDBC Driver Types/ Types of DriverManager

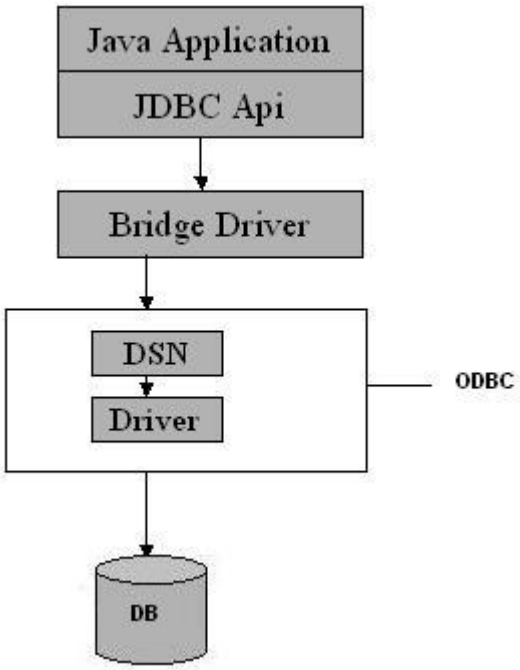
There a 4 types of drivers, which a Java application can use to access a relational database.

Given below are 4 types of JDBC drivers.

- ⇨ Type 1: JDBC-ODBC Bridge.
- ⇨ Type 2: Native API-partly Java Driver.
- ⇨ Type 3: JDBC-Net protocol Diver.
- ⇨ Type 4: Native-protocol pure Java Driver.

Type 1: JDBC ODBC Bridge

- JDBC-ODBC Bridge enables JDBC to access relational database via ODBC driver.
- The term "Bridge" clearly indicates that this driver acts as a connecting medium between JDBC & ODBC.
- ODBC (Open DataBase Connectivity) predates JDBC & is widely used to access database in non-Java environment.
- The driver is included in package sun.jdbc.odbc.



- Here java statements are converted to JDBC, JDBC statements calls ODBC by using JDBC-ODBC Driver.
- Driver translates all the code to ODBC, ODBC calls database & respond back.
- ODBC is a generic API.

ADVANTAGES:

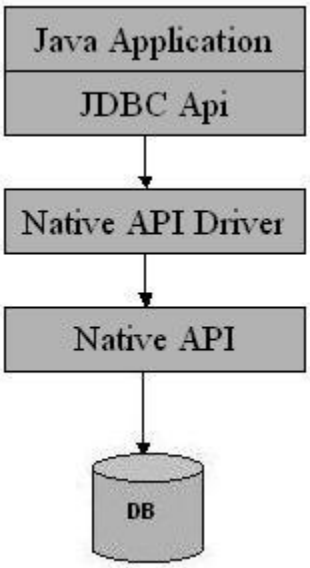
- It offers the ability to connect to all the databases on almost every platform.
- It is the only way to acess low-end desktop database & application.

DISADVANTAGE:

- ODBC drivers must be loaded on the target machine.
- Translation between JDBC & ODBC affects performance.

Type 2: Native API-partly Java Driver.

- Type 2 driver use Java Native Interface (JNI) to make calls to local database library API.
- Java native methods are used to invoke the API function that performs database operation.



- The driver converts JDBC method calls into native calls of database API.
- Type 2 driver converts the JDBC calls according to the specified database.
- Type 2 driver is not entirely written in java as it interfaces with non-Java code that makes the final database call.

  Eq. call for Oracle API

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

For MySQL Database.

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

Advantage:

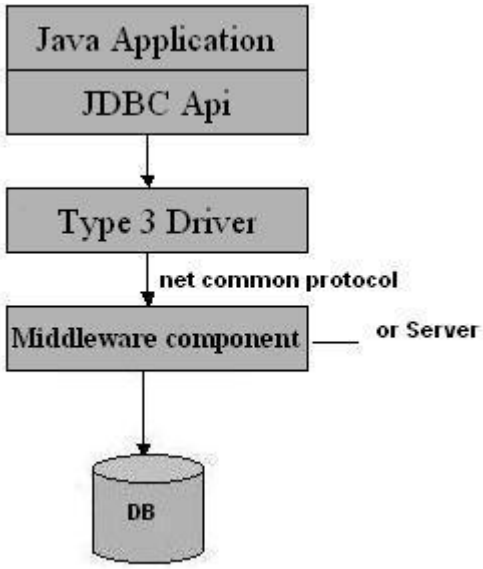
- Type 2 Driver is faster than Type ! Driver.
- Type 2 Driver is database specific.

Disadvantage:

- Type 2 Driver requires native code on target machine.
- It is not portable.
- Type 2 Drivers are not thread safe.

Type 3:JDBC-NET PROTOCOL DRIVER

- Type 3 Drivers are pure Java Drivers.
- Type 3 Driver, translates JDBC calls into a DBMS independent net protocol.
- DBMS independent net protocol, and then communicates with JDBC middleware component on server.



- Middleware component translates the network protocol to database specific function call.
- Type 3 JDBC Drivers are the most flexible JDBC solution, because they do not require any code on the client.
-

- Hence, as no Native API or libraries are in scenario, these drivers can be used to connect many databases on backend.

Advantage:

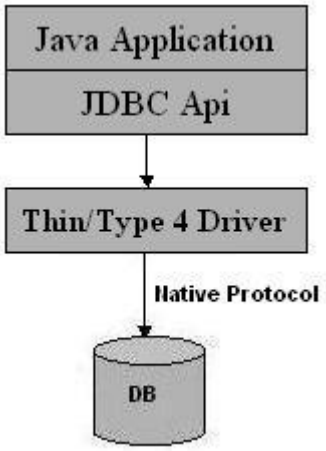
- No native API is required.
- No client side installations are required.
- It is portable & scalable.
- Driver is very flexible & allows access to multiple databases.

Disadvantage:

- Major drawback of Type 3 driver is that it have a very complicated architecture.

Type 4: NATIVE- Protocol pure Java Driver.

- Type 4 Driver is a native protocol, 100% Java driver.
- This allows direct calls from java client to DBMS server.
- As Type 4 Drivers are written in Java, it requires no configuration on client machine other than telling application where driver is.



Advantage:

- Driver is 100% platform independent.
- Eliminates deployment administration issues.
- No client-side installation.
- Good Performance.

Disadvantage:

- Requires a different driver for each database.

Seven Steps to Access Database using JDBC API.

Step 1: **Import java.sql package.**

Step 2: **Load & Register the JDBC Driver.**

Step 3: **Establish a Connection to Database Server.**

Step 4: **Create a Statement Object.**

Step 5: **Execute a Query using Statement Object.**

Step 6: **Retrieve Data from returned ResultSet Object.**

Step 7: **Close the Statement & Connection.**

Step 1: Import java.sql package.

- java.sql package contains all those classes & interfaces . which are required by JDBC API.
- This package contains all those base classes & interfaces which are used to property instantiate JDBC Drivers.

```
import java.sql.*;
```

Step 2: Load & Register the JDBC Driver.

Loading Driver:

- Drivers are exlicitly loaded using class java.lang.Class.
- Loading is performed using static method forName() of java.lang.Class.
- Signature of method forName() is.
  public static Object forName(String className)
- The method instantiate & loads the driver, we wish to use in our program.

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

Registering Driver:

- All the drivers are needed to be registered with java.sql.DriverManager, so that DriverManager uses correct driver at time of connetion.
- But registration of the driver is implicitly done by interface java.sql.Driver interface.

```
DriverManager.registerDriver();
```

Step 3: Establish Connection with Database Server.

- For establishing a connection with database server, we use interface java.sql.Connection.
- For acquiring a connection, static method of java.sql.Connection is used i.e.getConnection().
- Method getConnection() takes an URL-Uniform Resource Loader, which is usually termed as JDBC URL.
- **jdbc:<sub-protocol>:<sub-name>**
- Here JDBC is the standard base.
- Sub-protocol is the particular protocol type for eq. Odbc, Oracle, MySql, etc.
- Sub_Name is the additional specification used by sub_Protocol.
- JDBC URL is used by JDBC Driver to identify themselves & connect to specified database.
- Sub_Name is majorly the data source name or the connection URL to Database.

  While working with DSN i.e Type 1 Driver.

```
Connection con=DriverManager.getConnection("jdbc:odbc:DSN");
```

  While working with Type 4 Driver

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

Step 4: Create a Statement Object

- java.sql.Statement object acts as a container for executing SQL statement on given connection.
- For query execution, statement object is required.
- Statement object is created using createStatement() of Connection interface.

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

Step 5: Execute Query Using Statement Object.

- After creating a statement object, query has to be executed.
- Methods according to the query type is used, for eq.
  DRL- executeQuery() is used.
  DDL- execute() is used.
  DML- executeUpdate() is used.
- Now here, query is executed using above given methods of Statement class.

```
String query="Select * From STUDENTS;";
ResultSet rs=stmt.executeQuery(query);
```

Step 6: Retrieve the Data From returned ResultSet Object.

- java.sql.ResultSet object controls the access to the result returned by SQL Statements.
- ResultSet stores the returned data in tabular format, with its cursor placed on row before the first row.
- Later data is retrieved using getter methods of ResultSet object.

```
while(rs.next())
```

```
{
        String name=rs.getString(1);
        int id=rs.getInt(2);
}
```

Step 7: Close ResultSet & Connection.

- Once data is retrieved, ResultSet & Conection objects are closed using close() method.

```
rs.close();

con.close();
```

DEMO PROGRAM

```
import java.sql.*;

class  SelectDemo

{

        public static void main(String[] args)

        {

                try

                {

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

```
        Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/tech",”root”,”tech
”);
                        Statement stmt=con.createStatement();
                        String query="Select * From Product;";
                        ResultSet rs=stmt.executeQuery(query);


                        while (rs.next())
                        {
                                int id=rs.getInt(1);
                                String name=rs.getString(2);
                                String company=rs.getString(3);
                                int price=rs.getInt(4);


        System.out.println(""+id+"\t"+name+"\t"+company+"\t"+price);
                        }
                        stmt.close();
                        con.close();
                }
                catch (Exception e)
                {
```

```
            System.out.println("Exception Occured : "+ e.printStackTrace());
            }
        }
}
```

Advantage or Disadvantage of using JDBC or ODBC

- ODBC stands for Open DataBase Connectivity.
- ODBC is a Microsoft's product.
- ODBC's goal is to provide connectivity to hetrogeneous databases.
- JDBC stands for Java Database Connectivity.
- JDBC is a Sun MicroSystems product.
- JDBC's goal is to enable a java program to connect with a database.
- ODBC can make connectivity to any database, hense it is LANGUAGE INDEPENDENT.
- JDBC is used to write a single database application that can run on different platforms & interacts with different databases, hence JDBC is LANGUAGE DEPENDENT.
- ODBC is PROCEDURE ORIENTED
- JDBC is OBJECT ORIENTED.
- ODBC works only on Microsoft's platform.

- As JDBC API is written I java, it works on any platform, hence it is PLATFORM INDEPENDENT.
- ODBC mixes simple & advanced feature together for simple queries.
- JDBC is designed to achieved advanced feature but keeping interface as simple as possible.
- ODBC requires manual installation of ODBC Driver on all clients machine.
- JDBC Drivers are written in java & JDBC code is automatically installable, portable & secure on all platforms.

DriverManager

- DriverManager is a class in java.sql package.
- java.sql.DriverManager provides basic service for managing a set of JDBC Drivers.
- While initialization, DriverManager makes an attempt to load the drivers referenced in the "jdbc.drivers" system property.
- Explicit loading of Driver can also be done using forName() method of class java.lang.Class

  Class.forName("com.mysql.jdbc.Driver");
- This property allows the user to customize their application.
- Every driver which is newly loaded should register itself with DriverManager class using method registerDriver().
- Usually drivers does this itself.
- Main job of Jdbc DriverManager is to choose the appropriate Driver which is loaded to access the database.
- DriverManager a full list of loaded driver & just put the correct driver at correct place.
- DriverManager is the backbone of JDBC Architecture.
- Whenever method getConnection() is invoked, DriverManager attempts to locate a suitable driver from those loaded implicitly at initialization and from those which are loaded exliplicitly using JDBC URL.
- JDBC DriverManager is responsible for getting connection, as method getConnection() returns java.sql.Connection.

  Connection con= DriverManager.getConnection("jdbc:odbc:DSN");

- DriverManager not only manager drivers, but it also manager login time, login writer, printing traces and it does all this using it's methods.

Methods

- **public static connection getConnection(String url) throws SQLException**
  - ✓ Attempts to establish a Connection to given database URL.
  - ✓ To get the connection, DriverManager attempts to select an appropriate driver from a set of registered Jdbc driver.

- **public static Connection getConnection(String url, String user,String password)**
  - ✓ Attempts to establish a Connection to given database URL.
- **public static void registerDriver(Driver driver) throws SQLException**
  - ✓ Registers the given driver with DriverManager.
- **public static void deregisterDriver(Driver driver)throws SQLException**
  - ✓ Drops the driver from DriverManager list.
- **public static Driver getDriver(String url) throws SQLException**
  - ✓ Attempts to locate a driver that understand a given URL. URL-JDBC URL of form jdbc:subprotocol:subname.
- **public static void setLoginTimeout(int seconds)**
  - ✓ Sets the maximum time in seconds that a driver will wait while attempting to connect a database.
- **public static int getLoginTimeOut()**
  - ✓ Gets the maximum time in seconds that a driver can wait when attempting to login to a database.
  - ✓ All the above method contribute to the working of DriverManager class.
  - ✓ Basic working we can say from above methods is that DriverManager tries to locate a Driver & establish connection with the database.

Connection

- Connection is an interface in java.sql package.
- Connection interface represents session with a specific database.
- Connection gained with the database using java.sql.Connection interface is able to describe it's information about the tables.
- By default Connection Object is in auto-commit mode, which means that query executed will directly make the changes in the database table.
- Connection Object is gained using method getConnection() with specified database on correct Driver.
  Connection con= DriverManager.getConnection("jdbc:odbc:DSN");
- The connection object provides connection between the JDBC API & DBMS.
- java.sql.Statement acts as a container for SQL queries & to get the statement object, Connection's method are used.
- It is but obvious that statement will be created on established connection object.
  - ✓ java.sql.Statements.
  - ✓ java.sql.PreparedStatements.
  - ✓ java.sql.CallableStatement.
- Connection object is also used to gain information about the database i.e. metadata of database.
- Statement created using Connection object is created as per the requirement.
  - ✓ If normal execution has to be done, we use createStatement() method.
  - ✓ If repeated execution is to be done, we use prepareStatement() method.
  - ✓ If stored produces are to be called prepareCall() method is used.

Methods:

- **public Statement createStatement()**
  - ✓ Creates a statement object for sending SQL Statements to the database.
  - ✓ SQL Statement without parameter are executed normally using Statement object.
- **public PreparedStatement prepareStatement(String sql)**
  - ✓ Creates a PreparedStatement object for sending parameterized SQL statements to the database.
  - ✓ SQL statement with or without IN parameter(?) can be precompiled & stored in a PreparedStatement object.

- **public CallableStatement prepareCall(String sql)**
  - ✓ Creates a CallableStatement object for calling database stored procedures.
  - ✓ CallableStatement provides various methods for executing the call to the stored procedure.
- **public void commit()**
  - ✓ Makes all the changes mode.
  - ✓ Method should be used only when auto-commit mode has been disabled.
- **public void rollback()**
  - ✓ Undoes all the changes made in current transaction.
  - ✓ Method should be used only when auto-commit mode has been disabled.
- **public DatabaseMetaData getMetaData()**
  - ✓ Returns a DatabaseMetaData object that contains metadata about the database to which Connection object represents a connection.
- **public void close()**
  - ✓ Release this Connection object's database & JDBC resource immeidiately.

Statement

- Statement is an interface in java.sql package.
- Statement object acts as a container for executing a SQL statement on a given Connection.
- Statement object is used for executing static SQL statements & returning the result it produces.
- A single Statement object can have only one ResultSet object assigned, hence every new statement has to be used.
- JDBC has no implication on type of SQL statement passed to the DBMS, hense it is very flexible in it's statement part.
- For executing simple SQL statement java.sql.Statement object is used & the object is created using method createStatement() on specified Connection.
- As known type of SQL commands, there are also different types of methods for it's execution.
- DDL i.e. Data Definition Languages is handled using execute() method, which executes SQL statement that may return multiple results.
- DML i.e. Data Manipulation Language is handeled by executeUpdate(String sql) method which executes SQL INSERT, UPDATE or DELETE statement, which returns the numbers of rows affected.
- DRL i.e. Data Retrival Language is handeled by executeQuery(String url) method which executes SQL Statements which returns a single ResultSet.
- All the execution methods are called on statement object & they return the result appropriately.
- Even a bath SQL statement can be executed using executeBatch() method which returns an integer array of number of rows affected.

Methods:

- **public boolean execute(String sql)**
  - ✓ Executes the given SQL statement, which may return multiple results.
  - ✓ Returns true if first result is ResultSet object else returns false if result is an update Count.
- **public ResultSet executeQuery(String sql)**
  - ✓ Executes the given SQL statement , which returns a single ResultSet object.
- **public int executeUpdate(String sql)**
  - ✓ Executes the given SQL statements, which may be an INSERT, UPDATE or DELETE statement which may not return anything.

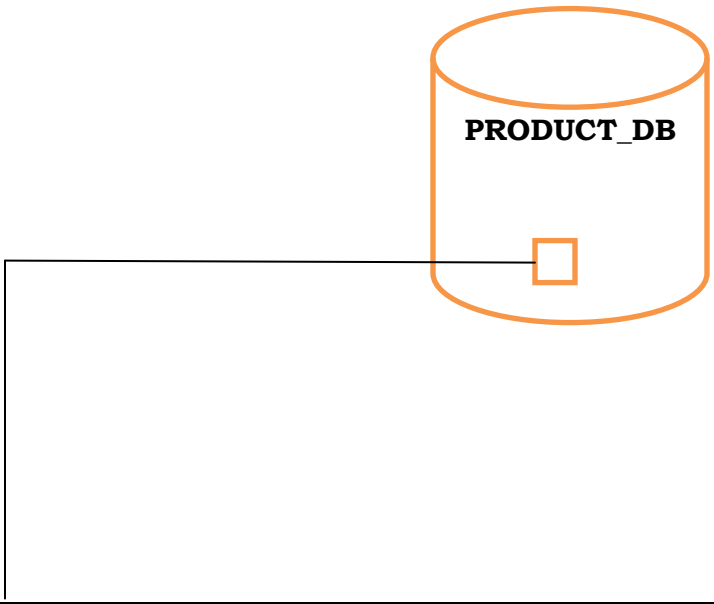
- **public void close()**
  - ✓ Releases this Statement object's database & JDBc resources immediately.
- **public void cancel()**
  - ✓ Cancels this Statement objects if both DBMS & Driver supports aborting an SQL Statement.

ALL THE PROGRAMS ILLUSTRATED ARE ACCORDING TO:

Database: PRODUCT_DB

Table: Product

DSN: ProductDSN



Product

P_id	P_name	Company	Price
1	N8	Nokia	24000.00
2	5830	BlackBerry	32000.00
3	Galaxy Ace	Samsung	14000.00

4	Xperia	Sony	25000.00
5	WildFire	HTC	23000.00

Programs

CreateTableDemo.java

```
import java.sql.*;


class  CreateTableDemo

{

	public static void main(String[] args)

	{

		try

		{

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

Connection con;
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/product","r
ooot","tech");
```

```
            Statement stmt=con.createStatement();
        String query="Create Table Product (p_id int primary key,   p_name
        varchar(50),company varchar(50),price money);";
                    stmt.execute(query);
                    System.out.println("Table created Successfully");


                    stmt.close();
                    con.close();
                }
                catch (Exception e)
                {
                    System.out.println("Exception Occured : "+ e);
                }
        }
}
```

InsertDataDemo.java

```
import java.sql.*;


class  InsertDataDemo

{

	public static void main(String[] args)

	{

		try

		{

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

Connection con;
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/product","r
ooot","tech");

		Statement stmt=con.createStatement();

		String query="Insert into Product
values("+1+",'N8','Nokia',"+24000+");";

			int rA=stmt.executeUpdate(query);

			System.out.println("Number Of Rows Affected :"+rA);

			stmt.close();

			con.close();

		}
```

```
            catch (Exception e)
            {
                  System.out.println("Exception Occured : "+ e);
            }
      }
}
```

UpdateDataDemo.java

```
import java.sql.*;
class  UpdateDataDemo
      {
            public static void main(String[] args)
            {
                  try
                  {
      Class.forName("com.mysql.jdbc.Driver");
Connection con;
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/product","r
ooot","tech");
                  Statement stmt=con.createStatement();
                  String query="Update Product Set price=23000 where
      p_name like 'wildfire'";
                  int rA=stmt.executeUpdate(query);
                  System.out.println("Number Of Rows Affected :"+rA);
```

```
                stmt.close();
                con.close();
                }
                catch (Exception e)
                {
                        System.out.println("Exception Occured : "+ e);
                }
        }
}
```

DeleteDataDemo.java

```
import java.sql.*;

class  DeleteDataDemo
{
        public static void main(String[] args)
        {
                try
                {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection
con=DriverManager.getConnection("jdbc:odbc:ProductDSN");
                        Statement stmt=con.createStatement();
                        String query="Delete From Product where company
like 'sony'";
                        int rA=stmt.executeUpdate(query);
                        System.out.println("Number Of Rows Affected
:"+rA);
```

```
                    stmt.close();
                    con.close();
                }
                catch (Exception e)
                {
                    System.out.println("Exception Occured : "+ e);
                }
            }
}
```

PreparedStatement

- Prepared Statement is an interface in java.sql package.
- PreparedStatement is sub-interface of java.sql.Statement.
- PreparedStatements represents a precompiled SQL statements.
- java.sql.Statements acts as a container for SQL Query, which is to be executed.
- PreparedStatement stores the same, but here queries are precompiled.
- PreCompilation means these statements can be executed more efficiently than the normal one.
- Whenever execute(), executeQuery() or executeUpdate() is invoked using Statement interface, it get compiled every time a new query interacts with backend.
- It indicates that, every new query is a fresh query& get compiled each time.
- PreparedStatement supports a concept of IN parameters, which acts a place holders for variables.
- IN parameter is denoted using ?.
- This parameter are set using setter methods.
- A typical setter method looks like
  public void setObject(int parameterIndex, Object value) throws SQLException
- PreparedStatement is generally used when a program is supposed to insert 100 records into a table or to select something frequently from a table.
- PreparedStatement is created using method prepareStatement() of Connection Interface.
  PreparedStatement prepareStatement(String sql) throws SQLException
- ResultSet created using PreparedStatement will be of type **TYPE_FORWARD_ONLY** & concurrency level of **CONCUR_READ_ONLY.**
- In give example can represents an active connection.
  String query= ”Select * from Products where P_ID=?”;
  PreparedStatement pstmt=con.prepareStatement(query);
  pstmt.setInt(1,5);
  Here, First parameter index of IN parameter, while second is the value provided to the corresponding parmeter.

Methods:

- **public boolean execute()**

- ✓ Executes the SQL Statement in this PreparedStatement object, which may be any kind of SQL Statement.

- **public ResultSet executeQuery()**
  - ✓ Executes the SQL query in this PreparedStatement object & returns the ResultSet object generated by the query.
- **public int executeUpdate()**
  - ✓ Executes the SQL statements in this Prepared Statement object, which mustbe an Data Manipulation Language(DML) statement such as insert, update or delete or DDL statement which returns nothing.
- **public void setObject(int parameter, Object x)**
  - ✓ Sets the value of designated parameter,using the given object.
  - ✓ Second parameter must be of type Object, hence equivalent object should be passed for it.
- **public ResultSetMetaData getMetaData()**
  - ✓ Retrives a ResultSetMetaData object that contains the information about columns of ResultSet object that will be returned when this PreparedStatement object is executed.

Demo Program

```
import java.sql.*;

class PreparedStatementInsertDataDemo
{
	public static void main(String[] args)
	{
		try
		{

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

```
Connection con;
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/product","r
ooot","tech");

String query="Insert into product values(?,?,?,?);"

                    PreparedStatement pstmt=con.prepareStatement(query);
                        pstmt.setInt(1,6);
                        pstmt.setString(2,"F9");
                        pstmt.setString(3,"Motorola");
                        pstmt.setInt(4,12000);

                        int rA=pstmt.executeUpdate(query);
                    System.out.println("Number Of Rows Affected :"+rA);
                        pstmt.close();
                        con.close();
                    }
                    catch (Exception e)
                    {
                        System.out.println("Exception Occured : "+ e);
                    }
            }
}
```

Command Line Argument Example

```
    import java.sql.*;

    class  InsertDataCLADemo
    {
```

```
        public static void main(String[] args)
        {
                int id=Integer.parseInt(args[0]);
                String name=args[1];
                String company=args[2];
                int price=Integer.parseInt(args[3]);
                try
                {

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

Connection con;
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/product","r
ooot","tech");

                        String query="Insert Into Product values(?,?,?,?)";
                PreparedStatement pstmt=con.prepareStatement(query);
                        pstmt.setInt(1,id);
                        pstmt.setString(2,name);
                        pstmt.setString(3,company);
                        pstmt.setInt(4,price);

                        int rA=pstmt.executeUpdate(query);
                        System.out.println("Number Of Rows Affected
    :"+rA);
                        pstmt.close();
                        con.close();
                }
                catch (Exception e)
                {
```

```
                    System.out.println("Exception Occured : "+ e);
                }
            }
        }
```

CallableStatement

- A java.sql.CallableStatement object is returned by the prepareCall() method of Connection interface.
- We use the CallableStatement object for calling the procedures on the database.
- Procedures are pre-defined sequence of SQL commands that can be invoked when function defined by the stored procedures are to be carried out.
- The argument to the prepareCall() methodis a String object that will enable the driver to determine that this is not an ordinary SQL statement and need to be transformed into a form that will be understood by the database system.
- The simplest way to call a stored procedure is

  CallableStatement cstmt=con.prepareCall("{call app2}");

  Here app2 is a stored procedure.
- Stored procedures are already in backend we just have to call them.
- Application will give an error in case if it fails to find the procedure.
- After creating CallableStatement , executeQuery() or executeUpdate() is invoked.
- Output is stored in ResultSet object and data can be retrieved using getter methods.
- Stored Procedures can have arguments that specify input values (called IN parameter) to the operation.
- In this case parameter list between parentheses following the procedure name.
- Each parameter is denoted by a ? .
- Values for parameter are set using setter methods.

EXAMPLE

```
CallableStatement cstmt=con.prepareCall("{call getMonthData(?,?)}");

cstmt.setInt(1,6);

cstmt.setInt(2,1999);

ResultSet rs=cstmt.executeQuery();
```

- Procedures can also have parameters for returning result, also referred as OUT parameters and they are also indicated using ?.
- For each OUT parameter output values are identified by one of the types defined in java.sql.Types.
- They are registered using registerOutParameter() of CallableStatement.

  cstmt.registerOutParameter(2,Types.INTEGER);

- First argument in the list is index of OUT parameter and second is the type.

ResultSet

- ResultSet is an interface in java.sql package.
- ResultSet is a table of data representing a database resultset.
- This ResultSet is usually generated by the query executed.
- ResultSet's are arranged as a table, with column heading & values returned in the order specified in the statement.
- ResultSet object maintains a cursor pointing to its current row of data.
- Intially the cursor is positioned before the frist row.
- Cursor can be imagined as a pointer to the rows of ResultSet that has ability to keep track of which row is being currently accessed.
- User can access the data in a ResultSet using cursor one row at a time from top to bottom & bottom to top.
- By default ResultSet object is not updatable & has a cursor that moves forward only.
- Type of cursor can be specified using 3 fields.
  - ✓ **TYPE_FORWARD_ONLY**
  - ✓ **TYPE_SCROLL_INSENSITIVE**
  - ✓ **TYPE_SCROLL_SENSITIVE**

  1. **TYPE_FORWARD_ONLY**
  - ➢ Indicates cursor is not type scrollable & can move from before first row to after last row.

  2. **TYPE_SCROLL_INSENSITIVE**
  - ➢ Indicates ResultSet is type scrollable, but is not sensitive to change to data that underlies ResultSet.

  3. **TYPE_SCROLL_SENSITIVE**
  - ➢ Indicates ResultSet is type scrollable & is sensitive to changes to the data that underlies the ResultSet.

  4. Also concurrency mode for the ResultSet can be set using 2 fields.
     - **CONCUR_READ_ONLY**
     - **CONCUR_UPDATABLE**

- **CONCUR_READ_ONLY**
  Indicates that concurrency mode for ResultSet object that may Not be updated.
- **CONCUR_UPDATABLE**
  Indicates the concurrency mode for a ResultSet object that can be updated.
  ResultSet provides methods for scrolling up & down with positions relative & absolute of the rows accesed.

ResultSet also provides getter methods for retrieving column values from the current row.

Methods:

- **public boolean first()**
  Moves the cursor to the first row in this ResultSet object.
- **public boolean last()**
  Moves the cursor to the last row in this ResultSet object.
- **public void beforeFirst()**
  Moves the cursor to just before the first row.
- **public void afterLast()**
  Moves the cursor to the end of ResultSet object, just after the last row.
- **public boolean next()**
  Moves the cursor forward one row from it's current position.
- **public boolean previous()**

  Moves the cursor to the previous row in this ResultSet object.

- **public boolean relative(int rows)**
  Moves the cursor a relative number of rows either negative or positive.
- **public boolean absolute(int row)**
  Moves the cursor to the given number of row in ResultSet.
- **public Object getObject(String columnLabel)**
  Gets the value of designated column in the current ow of this ResultSet object.
- **public Object getObject(int columnIndex)**
  Retrives the value of designated column in the current row of this ResultSet object.
- **public ResultSetMetaData getMetaData()**
  Retrives the number, types & purpose of this ResultSet objects columns.
- **public void close()**
  Release this ResultSet objects database & JDBC resource.

Demo Program

```
import java.sql.*;
class InverseSelectDemo
{
    public static void main(String[] args)
```

```
        {
            try
    Class.forName("com.mysql.jdbc.Driver");
Connection con;
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/product","r
ooot","tech");
    Statement stmt;
    stmt=con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
                String query="Select * From Product";
                ResultSet rs=stmt.executeQuery(query);
                rs.last();
                while (rs.previous())
                {
                    int id=rs.getInt(1);
                    String name=rs.getString(2);
                    String company=rs.getString(3);
                    int price=rs.getInt(4);

        System.out.println(""+id+"\t"+name+"\t"+company+"\t"+price);
                }

                stmt.close();

                con.close();
            }
```

```
                catch (Exception e)
                {
                        System.out.println("Exception Occured : "+ e);
                }
        }
}
```

ResultSetMetaData

- ResultSetMetaData is an interface in java.sql package.
- ResultSetMetaData is an object that can be used to get information about the types & properties of the columns in ResultSet object.
- ResultSetMetaData is available by calling getMetaData() method of ResultSet.
- For eq.
  ResultSet rs=stmt.executeQuery("Select * from Product");
  ResultSetMetaData rsmd=rs.getMetaData();
- Now using thye various methods of ResultSetMetaData, all information about the ResultSet can be retrived.
- MetaData means information about the data.
- Hence ResultSetMetaData provides information about ResultSet.
  Methods:
  - **public int getColumnCount()**
    Returns the number of columns in this ResultSet.object.
  - **public String getColumnLabel(int column)**
    Gets the designated column's suggested title for use in printouts and displays.
  - **public String getColumnName(int column)**

    Get the designated column's name.

  - **public String getColumnTypeName(int column)**
    Retrives the designated column's database specific type name.
  - **public String getTableName(int column)**
    Get the designated column's table name.

Demo Program

```
import java.sql.*;

class ResultSetMetaDataDemo
{
    public static void main(String[] args)
    {
```

```
                try
Class.forName("com.mysql.jdbc.Driver");
Connection con;
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/product","r
ooot","tech");
Statement stmt;
stmt=con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,Result
Set.CONCUR_UPDATABLE);
                        String query="Select * From Product;";
                        ResultSet rs=stmt.executeQuery(query);
                        ResultSetMetaData rsmd=rs.getMetaData();

                        int cC=rsmd.getColumnCount();
                        System.out.println("Total Column(s) : "+cC);

                        for (int i=1;i<=cC ;i++ )
                        {
                                String cN=rsmd.getColumnName(i);
                                System.out.println("Column "+i+" "+cN);
                        }

                        stmt.close();

                        con.close();
                }
                catch (Exception e)
                {
```

```
                    System.out.println("Exception Occured : "+ e);
                }
            }
}
```