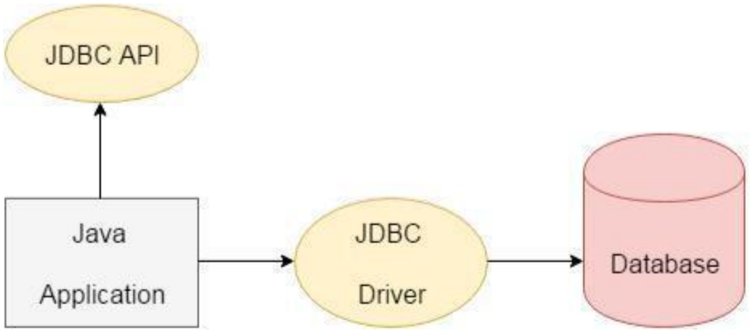
Database Programming

The Design of JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



Uses of JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses **ODBC driver which is written in C language** (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses **JDBC drivers (written in Java language).**

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

The traditional client/server model has a rich GUI on the client and a database on the server (figure below). In this model, a JDBC driver is deployed on the client.

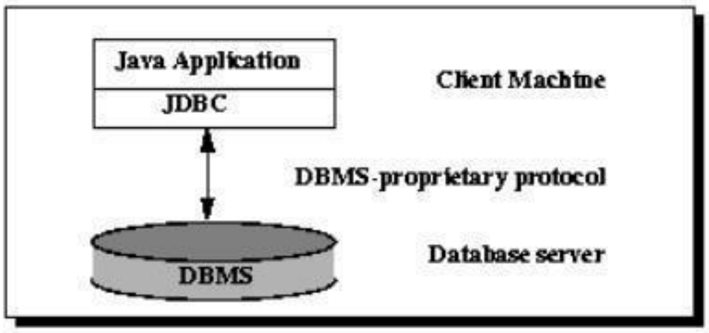


fig.Two-tier Architecture for Data Access.

However, the world is moving away from client/server and toward a "three-tier model" or even more advanced "n-tier models." In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates visual presentation (on the client) from the business logic (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java application or applet or a web form.

Communication between the client and middle tier can occur through HTTP (when you use a web browser as the client), RMI (when you use an application), or another mechanism. JDBC manages the communication between the middle tier and the back-end database. Figure below shows the basic three tier architecture.

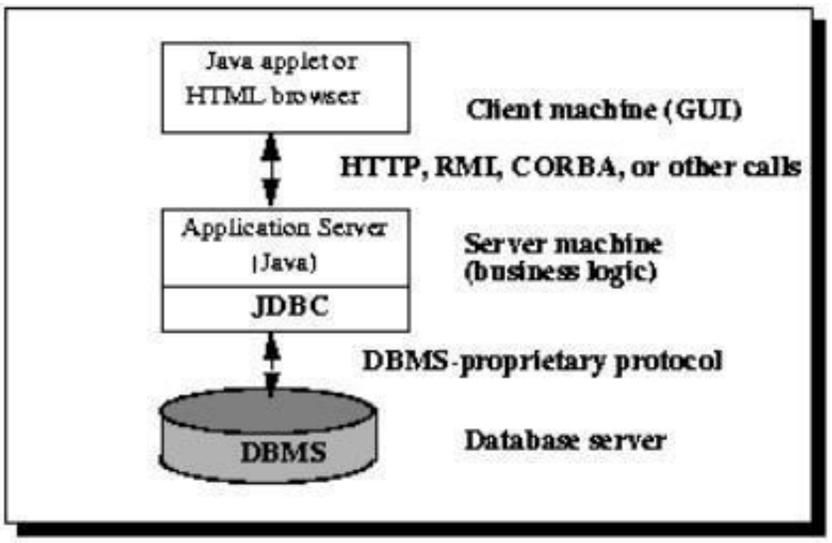

Java applet or HTML browser
Client machine (GUI)
HTTP, RMI, CORBA, or other calls
Application Server (Java)
JDBC
Server machine (business logic)
DBMS-proprietary protocol
DBMS
Database server

fig.Three-tier Architecture for Data Access

<u>JDBC Driver</u>

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

1) <u>JDBC-ODBC bridge driver</u>

The 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. This is now discouraged because of thin driver.

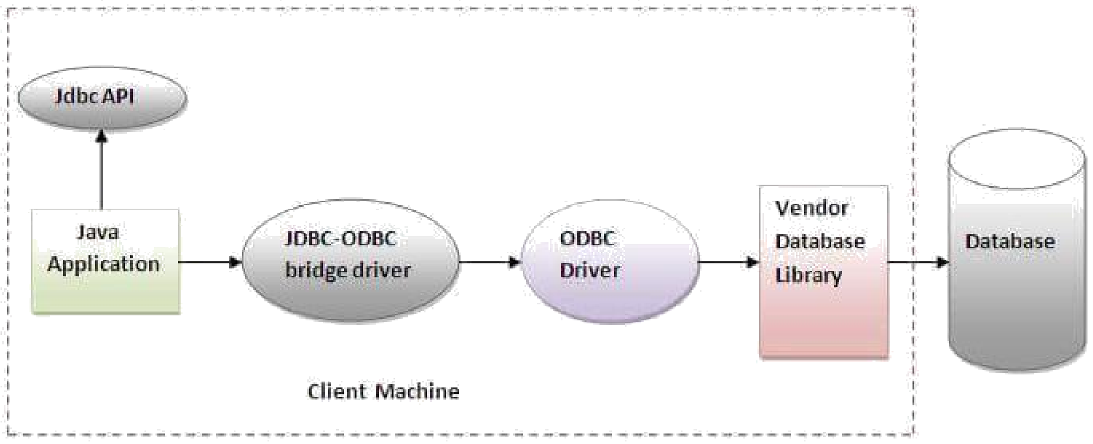


Figure- JDBC-ODBC Bridge Driver

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

2) <u>Native-API driver</u>

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. **It is not written entirely in java**.

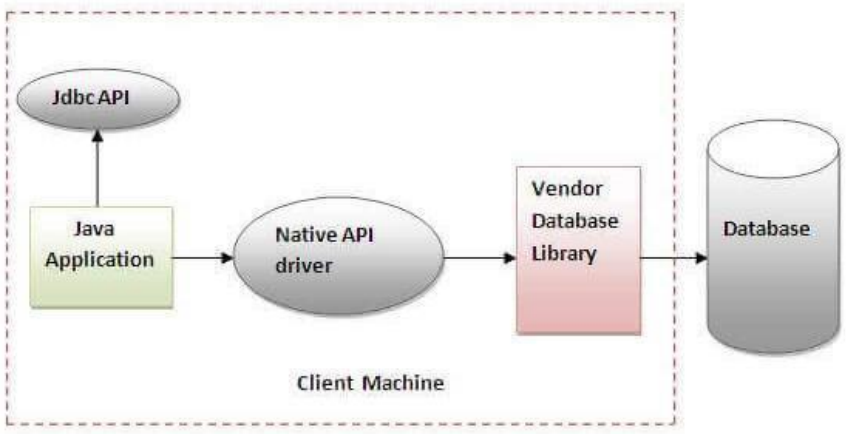


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine. o

The Vendor client library needs to be installed on client machine.

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

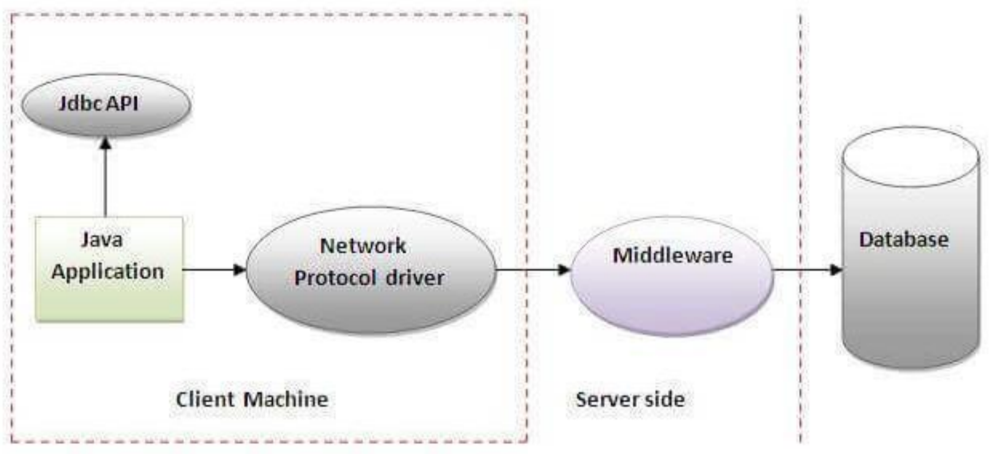


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

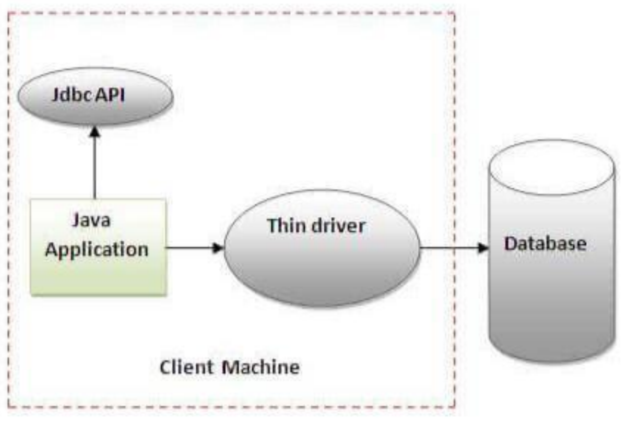


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

Using JDBC Drivers

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

1) Register the driver class

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Example to register the OracleDriver class

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

Example to register MySQL driver class

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

2) Create the connection object

The **getConnection()** method of DriverManager class is used to establish connection with the database.

Establish connection with the Oracle database

```
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","username","password");
```

Establish connection with the MySQL database

```
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/dbname","username","password");
```

3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

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

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

```
stmt.execute(query);
```

For Selecting Data

```
ResultSet rs=stmt.executeQuery(query);
while(rs.next()){
    //accessing data
    //rs.getInt(1) rs.getString(2)…
}
```

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

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

Creating Connection in Derby Database

Apache Derby is a Relational Database Management System which is fully based on (written/implemented in) Java programming language. It is an open source database developed by Apache Software Foundation.
Oracle released the equivalent of Apache Derby with the name JavaDB.

Derby provides an embedded database engine which can be embedded in to Java applications and it will be run in the same JVM as the application. Simply loading the driver starts the database and it stops with the applications.

Connecting to a Derby Database

Using Embedded Driver:

```
//Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
//optional for java 5.0 +

String dburl = "jdbc:derby:dbname;create=true"
Connection conn = DriverManager.getConnection(dburl);
if (conn != null) {
      System.out.println("Connected to database !");
}
```

Using Derby Client Driver:

```
//Class.forName("org.apache.derby.jdbc.ClientDriver");
    //optional for java 5.0 +
String dbURL2 = "jdbc:derby://localhost/webdb2;create=true";
            String user = "username";
            String password = "password";
            Connection conn2 = DriverManager.getConnection(dbURL2, user,
                                 password);
            if (conn2 != null) {
                System.out.println("Connected to database #2");
            }
```

Download JDBC Driver JAR Files:

Download jar files of related database.

Connecting to a SQLite Database

SQLite is a simple, small, fast, reliable, server-less, zero- configuration and no-installation SQL database library which is running in-process with the client application.

```
 Class.forName("org.sqlite.JDBC"); String
dburl="jdbc:sqlite:dbname"; Connection
conn=DriverManager.getConnection
                    (dburl);
 if (conn != null) {
       System.out.println("Connected to database !");
 }
```

Connecting to a MySQL Database

```
 Class.forName("com.mysql.jdbc.Driver");
 String dburl="jdbc:mysql://localhost:3306/dbname";
Connection conn=DriverManager.getConnection
 (dburl,"username","password"); if (conn !=
null) {
       System.out.println("Connected to database !");
 }
```

Connecting to a Oracle Database

```
Class.forName("oracle.jdbc.driver.OracleDriver");
String dburl=" jdbc:oracle:thin:@localhost:1521:xe";
Connection conn=DriverManager.getConnection
(dburl,"username","password"); if (conn !=
null) {
      System.out.println("Connected to database !");
}
```

Complete CRUD Operation using Derby Database

```
import java.sql.*;
public class Example {
    public static void main(String[] args)
     { String sql="";
     ResultSet rs;
     try {
           //connecting derby database
           String dburl="jdbc:derby:mydb;create=true";
           Connection conn=DriverManager.getConnection(dburl);
           System.out.println("Database connected");

           Statement st=conn.createStatement();

           //creating a table
           /*sql="CREATE TABLE student(sid INT, name VARCHAR(30),
                  address VARCHAR(30))";
           st.execute(sql);
           System.out.println("Table Created Successfully!");*/

           //clearing data before insert
         sql="DELETE FROM student";
           st.execute(sql);

           //inserting data
           sql="INSERT INTO student(sid,name,address) VALUES
                   (1,'hari','Btm'), (2,'Ram','Ktm')";
           st.execute(sql);
           System.out.println("Data Inserted Successfully\n");

           //retrieving data
           System.out.println("Data Before Update and Delete");
           sql="SELECT * FROM student";
           rs=st.executeQuery(sql);
           System.out.println("Sid\t"+"Name\t"+"address");
           while(rs.next()) {
                 int sid=rs.getInt(1);
```

```
                String name=rs.getString(2);
                String address=rs.getString(3);
                System.out.println(sid+"\t"+name+"\t"+address);
           }

           //updating data
           sql="UPDATE student SET name='Hari',address='Ktm'
                  WHERE sid=1";
           st.execute(sql);
           System.out.println("\nData Updated Successfully");

           //deleting data
           sql="DELETE FROM student WHERE sid=2";
           st.execute(sql);
           System.out.println("Data Deleted Successfully\n");

           //retrieving data
           System.out.println("Data After Update and Delete");
           sql="SELECT * FROM student";
           rs=st.executeQuery(sql);
           System.out.println("Sid\t"+"Name\t"+"address");
           while(rs.next()) {
                int sid=rs.getInt(1);
                String name=rs.getString(2);
                String address=rs.getString(3);
                System.out.println(sid+"\t"+name+"\t"+address);
           }

      }catch(Exception ex) {
         System.out.println(ex.toString());
      }
   }
}
```

Complete CRUD Operation using MySQL Database

```
import javax.swing.*;
import javax.swing.table.*;
import java.sql.*;

public class Example {
      Connection conn;
      Statement st;
      ResultSet rs;
      String sql="";
      DefaultTableModel tmodel;

      //creating connection
```

```
public void getConnection(){
      try {
            Class.forName("com.mysql.jdbc.Driver");
            String dburl="jdbc:mysql://localhost:3306/sixth";
            conn=DriverManager.getConnection(dburl,"root","");

            //creating a table
                 sql="CREATE TABLE IF NOT EXISTS student(sid
               INT, name VARCHAR(30), address VARCHAR(30))";
            st=conn.createStatement();
            st.execute(sql);
            //System.out.println("Table Created!");

      }catch(Exception ex) {
            System.out.println(ex.toString());
      }
}

Example(){
      getConnection();
      //creating UI
      JFrame jframe=new JFrame("My Frame");
```

```
jframe.setSize(600, 250);
jframe.setLocationRelativeTo(null);
jframe.setLayout(null);
jframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JLabel lbl1=new JLabel("Student
Id:"); lbl1.setBounds(20, 12, 100,
10); jframe.add(lbl1);

JTextField txt1=new JTextField();
txt1.setBounds(120, 10, 150, 20);
jframe.add(txt1);

JLabel lbl2=new JLabel("Student Name:");
lbl2.setBounds(20, 55, 100, 10);
jframe.add(lbl2);

JTextField txt2=new JTextField();
txt2.setBounds(120, 50, 150, 20);
jframe.add(txt2);

JLabel lbl3=new JLabel("Student Address:
"); lbl3.setBounds(20,85,120,30);
jframe.add(lbl3);

JTextField txt3=new JTextField();
txt3.setBounds(120, 90, 150, 20);
jframe.add(txt3);

JButton insert=new JButton("Insert");
insert.setBounds(10, 140, 80, 20);
jframe.add(insert);

JButton update=new JButton("Update");
update.setBounds(100, 140, 80, 20);
jframe.add(update);

JButton delete=new JButton("Delete");
delete.setBounds(200, 140, 80, 20);
jframe.add(delete);

JButton view=new JButton("View");
view.setBounds(300, 140, 80, 20);
jframe.add(view);

//creating empty JTable
String cols[]= {"Sid","Name","Address"};
tmodel=new DefaultTableModel(cols,0);
JTable jt=new JTable(tmodel); JScrollPane
sp=new JScrollPane(jt); sp.setBounds(300,
10, 250, 100);
```

```
        jframe.add(sp);
        //later we can add new row as follows
        //tmodel.addRow(new Object[] {1,"Ram","Btm"});

        insert.addActionListener(e->{
              int id=Integer.parseInt(txt1.getText().toString());
              String name=txt2.getText().toString();
              String
              address=txt3.getText().toString(); try {
                    sql="INSERT INTO student (sid,name,address)
                     VALUES('"+id+"','"+name+"','"+address+"')";
                    st.execute(sql);
                    JOptionPane.showMessageDialog(null, "Data
                        Inserted Successfully");
              }
              catch(Exception ex) {
                    System.out.println(ex.toString());
              }
        });

        update.addActionListener(e->{
              int id=Integer.parseInt(txt1.getText().toString());
              String name=txt2.getText().toString();
              String
              address=txt3.getText().toString(); try {
                    sql="UPDATE student SET name='"+name+"',
                     address='"+address+"' WHERE sid='"+id+"'";
                    st.execute(sql);
                    JOptionPane.showMessageDialog(null, "Data
                        Updated Successfully");
              }catch(Exception ex) {
                    System.out.println(ex.toString());
              }
        });

        delete.addActionListener(e->{
              int id=Integer.parseInt(txt1.getText().toString());
              try {
                    sql="DELETE FROM student WHERE sid='"+id+"'";
                    st.execute(sql);
                    JOptionPane.showMessageDialog(null, "Data
                       Deleted Successfully");
              }catch(Exception ex) {
                    System.out.println(ex.toString());
              }
        });

        view.addActionListener(e->{
              try {
                    sql="SELECT * FROM student";
                    rs=st.executeQuery(sql);
```

```
                    //clearing JTable
                    tmodel.setRowCount(0);
                    while(rs.next()) {
                          //plotting data in JTable
                          tmodel.addRow(new Object[] {
                                      rs.getInt(1),
                                      rs.getString(2),
                                      rs.getString(3)
                                      });
                    }
              }catch(Exception ex) {
                    System.out.println(ex.toString());
              }
        });


        jframe.setVisible(true);

    }

    public static void main(String[] args) {
        new Example();
    }
}
```

Example Using MVC

StudentView.java

```
import javax.swing.*;
import javax.swing.table.*;
public class StudentView {
    public  JLabel  lbl1,lbl2,lbl3;  public
    JTextField  txt1,txt2;  public  JButton
    btn1,btn2;  //required  for  creating  a
    empty list DefaultTableModel tmodel;


     public StudentView() {
       JFrame jf=new JFrame("Student Form");
       jf.setSize(400, 300);
       jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       jf.setLayout(null);
       jf.setLocationRelativeTo(null);

       lbl1=new JLabel("Enter Sid:");
       lbl1.setSize(100, 30);
       lbl1.setLocation(20, 20);
       jf.add(lbl1);
```

```
        txt1=new JTextField();
        txt1.setSize(120, 30);
        txt1.setLocation(100, 20);
        jf.add(txt1);

        lbl2=new JLabel("Enter Name:");
        lbl2.setSize(100, 30);
        lbl2.setLocation(20, 60);
        jf.add(lbl2);

        txt2=new JTextField();
        txt2.setSize(120, 30);
        txt2.setLocation(100, 60);
        jf.add(txt2);

        btn1=new JButton("Save");
        btn1.setSize(100, 20);
        btn1.setLocation(50, 110);
        jf.add(btn1);

        btn2=new JButton("Display");
        btn2.setSize(100, 20);
        btn2.setLocation(160, 110);
        jf.add(btn2);

        //creating empty table with default table model
        String cols[]= {"Sid","Name"}; tmodel=new
        DefaultTableModel(cols,0); //0 rows
        JTable jt=new JTable(tmodel);
        JScrollPane jp=new JScrollPane(jt);
        jp.setLocation(50, 150);
        jp.setSize(200, 100);
        jf.add(jp);

        jf.setVisible(true);
      }
}
```

StudentModel.java

```
import java.sql.*;
public class StudentModel {
   private int sid;
   private String name;
   private Connection conn;
   private ResultSet rs;
   private Statement st;
   private String sql="";
```

```
    public void setId(int sid) {
       this.sid=sid;
    }

    public int getId() {
       return sid;
    }

    public void setName(String name) {
       this.name=name;
    }

    public String getName() {
       return name;
    }

   //creating connection
      public void getConnection(){
            try {
                  Class.forName("com.mysql.jdbc.Driver");
                  String dburl="jdbc:mysql://localhost:3306/sixth";
                  conn=DriverManager.getConnection(dburl,"root","");

                  //creating a table
                  String sql="CREATE TABLE IF NOT EXISTS student(sid
                        INT, name VARCHAR(30))";
                  st=conn.createStatement();
                  st.execute(sql);
                  //System.out.println("Table Created!");
            }catch(Exception ex) {
                  System.out.println(ex.toString());
            }
      }

      public void in_up_del() {
            try {
                  st=conn.createStatement();
                  sql="INSERT INTO student(sid,name) VALUES
                        ('"+sid+"','"+name+"')";
                  st.execute(sql);
            }catch(Exception ex) {
                  System.out.println(ex.toString());
            }
      }

      public ResultSet get_all_data() {
            try {
                  st=conn.createStatement();
                  sql="SELECT * FROM student";
                  rs=st.executeQuery(sql);
            }catch(Exception ex) {
                  System.out.println(ex.toString());
            }
            return rs;
      }
}
```

StudentController.java

```
import javax.swing.*;
import java.sql.*;
public class StudentController {
   StudentView v;
   StudentModel m;

   public void initController() {
         //initializing view
         v=new StudentView();
         m=new StudentModel();
         m.getConnection();
         //registering events
         v.btn1.addActionListener(e->saveClicked());
         v.btn2.addActionListener(e->displayClicked());
   }

   public void saveClicked() {
         int sid=Integer.parseInt(v.txt1.getText());
         String name=v.txt2.getText();
         m.setId(sid);
         m.setName(name);m.in_up_del();
         JOptionPane.showMessageDialog(null, "Saved Successfully!");
   }

   public void displayClicked() {
         //clearing all rows
         v.tmodel.setRowCount(0);
         ResultSet rs=m.get_all_data();
         try {
              while(rs.next()) {
                   Object[] obj= {rs.getInt(1),rs.getString(2)};
                   v.tmodel.addRow(obj);;
              }
         }catch(Exception ex) {}
   }
}
```

Example.java

```
public class Example {
     public static void main(String[] args) {
          StudentController cont=new
          StudentController(); cont.initController();
     }
}
```

Using Prepared Statement Interface

The Prepared Statement interface is a sub-interface of Statement. It is used to execute parameterized query.

Let's see the example of parameterized query:

```
String sql="insert into emp values(?,?,?)";
```

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

Syntax of Prepared Statement:

```
PreparedStatement pst=conn.prepareStatement(query);
```

The important methods of PreparedStatement interface are given below:

Method	**Description**
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
Public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
Public void setFloat(int paramIndex, float value)	sets the float value to the given parameter
Public void setDouble(int paramIndex, double value)	index. sets the double value to the given parameter index.
Pubilic int execute() **Public int executeUpdate()**	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

Inserting data using Prepared Statement

```
String sql="INSERT INTO student(sid,name)
VALUES(?,?)"; PreparedStatement
pst=conn.prepareStatement(sql); pst.setInt(1, 101);
pst.setString(2, "Ram");
pst.executeUpdate(sql); //pst.execute(sql);
```

Insert multiple rows dynamically using Prepared Statement

```
 String sql="INSERT INTO student(sid,name)
VALUES(?,?)"; PreparedStatement
pst=conn.prepareStatement(sql); //inserting five set
of records dynamically Scanner scan=new
Scanner(System.in); for(int i=1;i<=2;i++) {
       System.out.println("Enter Sid:");
       int sid=scan.nextInt();
      System.out.println("Enter Name:");
      String name=scan.next();

     pst.setInt(1, sid);
     pst.setString(2, name);
     pst.executeUpdate();
     System.out.println("Inserted Successfully!");
 }
```

Updating data using Prepared Statement

```
String sql="UPDATE student SET name=? WHERE sid=?";
PreparedStatement pst=conn.prepareStatement(sql);
pst.setString(1, "Raaju Poudel"); pst.setInt(2,
101);
pst.executeUpdate();
System.out.println("Updated Successfully!");
```

Deleting data using Prepared Statement

```
String sql="DELETE FROM student WHERE sid=?";
PreparedStatement pst=conn.prepareStatement(sql);
pst.setInt(1, 101);
pst.executeUpdate();
System.out.println("Deleted Successfully!");
```

Selecting data using Prepared Statement

```
String sql="select * from student";
PreparedStatement pst=conn.prepareStatement(sql);
ResultSet rs=pst.executeQuery();
while(rs.next()) {
    System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

Difference between Statement and Prepared Statement

Statement	PreparedStatement
It is used when SQL query is to be executed only once.	It is used when SQL query is to be executed multiple times.
You can not pass parameters at runtime.	You can pass parameters at runtime.
Used for CREATE, ALTER, DROP statements.	Used for the queries which are to be executed multiple times. Performance
Performance is very low.	is better than Statement. It extends
It is base interface.	statement interface.
Used to execute normal SQL	Used to execute dynamic SQL queries.
queries. We can not used statement for reading binary data.	We can used Preparedstatement for reading binary data.
It is used for DDL statements.	It is used for any SQL Query.
We can not used statement for writing binary data.	We can used Preparedstatement for writing binary data.
No binary protocol is used for communication.	Binary protocol is used for communication.

Storing image in database using Prepared Statement

Some important points about to handle the large objects in JDBC.

- If want to insert an image using JDBC into database, or read an image from database then we need to use PreparedStatement of JDBC
- In database, the image will not be stored directly. The bytes of an image (binary data) will be stored.
- To store the image in data base, we should declare the column type as ***BLOB***.
- BLOB type of column can store the data up-to a maximum of 4 GB.
- **To set the binary data (bytes) of an image to sql command, we need to use the setBinaryStream() method on PrepareStatement object.**

Example is shown below,

```
 import java.sql.*;
import java.io.*;
 import javax.swing.*;
 public class Example{
      public static void main(String[] args) {
            try {
                  Class.forName("org.sqlite.JDBC");
                  String dburl="jdbc:sqlite:testdb";
                  Connection conn=DriverManager.getConnection(dburl);

                  //create a table as follows
                  /*
                  String sql="CREATE TABLE imgtest(id INT,
                              name VARCHAR(30), image BLOB)";
                  PreparedStatement
                  pst=conn.prepareStatement(sql); pst.execute();
                  System.out.println("Table Created !");
                  */

                  //inserting data including image
                  String sql="INSERT INTO imgtest VALUES(?,?,?)";
                  PreparedStatement pst=conn.prepareStatement(sql);
                  pst.setInt(1, 101);
                  pst.setString(2, "Prakash Koirala");
                  //for setting image
                  String imgurl="/Users/Desktop/test.jpg"; File
                  file=new File(imgurl);
                  FileInputStream fstream=new FileInputStream(file);
                  pst.setBinaryStream(3, fstream,
                  fstream.available()); pst.executeUpdate();
                  System.out.println("Data Inserted
                  Successfully!"); conn.close();
            }catch(Exception ex) {
                  System.out.println(ex.toString());
            }

      }
 }
```

Displaying image from database in JFrame

```
import java.sql.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;
public class Example1 {
     public static void main(String[] args) {
           //creating JFrame
           JFrame jf=new JFrame("My Frame");
           jf.setSize(350, 300);
           jf.setLocationRelativeTo(null);
           jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
           jf.setLayout(null);

         try{
              Class.forName("org.sqlite.JDBC");
              String dburl="jdbc:sqlite:testdb";
              Connection conn=DriverManager.getConnection(dburl);

              //selecting data
              String sql="SELECT image FROM imgtest WHERE id=101";
              PreparedStatement pst=conn.prepareStatement(sql);
              ResultSet rs=pst.executeQuery();

              byte[] bytes=null;
              while(rs.next()) {
                    bytes=rs.getBytes(1);
              }

              //plotting image in JLabel
              Image image=Toolkit.getDefaultToolkit().createImage(bytes);
              image=image.getScaledInstance(180, 180,
                             Image.SCALE_SMOOTH);
              ImageIcon imgicon=new ImageIcon(image);
              JLabel lbl=new JLabel();
              lbl.setIcon(imgicon); lbl.setSize(180,
              180); lbl.setLocation(80, 50);
              jf.add(lbl);


        }catch(Exception ex) {
              System.out.println(ex.toString());
        }

         jf.setVisible(true);
     }
}
```

ResultSetMetaData Interface

The typical definition of a MetaData is data about the data. ResultSetMetaData means, the data about the ResultSet is called ResultSetMetaData.

It provides all necessary information about a ResultSet such as,

- Database Name
- Table Name
- Column Type
- Column Name
- Column Type Name
- Column Class Name

Syntax of using ResultSetMetaData,

```
ResultSet rs = stmt.executeQuery("sql");
ResultSetMetaData resultSetMetaData = rs.getMetaData();
```

Example is shown below:

```
import java.sql.*;
public class Example1 {
     public static void main(String[] args)
         { try{
               Class.forName("org.sqlite.JDBC");
               String dburl="jdbc:sqlite:testdb";
               Connection conn=DriverManager.getConnection(dburl);

               //selecting data
               String sql="SELECT * FROM imgtest WHERE id=101";
               PreparedStatement pst=conn.prepareStatement(sql);
               ResultSet rs=pst.executeQuery();

               ResultSetMetaData meta=rs.getMetaData();
               System.out.println("Database: "+meta.getCatalogName(1));
               System.out.println("Table: "+meta.getTableName(1));
               System.out.println("Column Name: "+meta.getColumnName(1));
               System.out.println("Column Type Name: "
                               +meta.getColumnTypeName(1));


               System.out.println("Column Name: "+meta.getColumnName(2));
               System.out.println("Column Type Name: "
                          +meta.getColumnClassName(2));

         }catch(Exception ex) {
               System.out.println(ex.toString());
         }

     }
}
```

Output:

```
Database: imgtest
Table: imgtest
Column Name: id
Column Type Name: INT
Column Name: name
Column Type Name: java.lang.Object
```

Scrollable and Updatable Result Set

Whenever we create an object of ResultSet by default, it allows us to retrieve in the forward direction only and we cannot perform any modifications on ResultSet object.
Therefore, by default, the ResultSet object is non-scrollable and non-updatable ResultSet.

JDBC Scrollable ResultSet :

A scrollable ResultSet is one which allows us to retrieve the data in forward direction as well as backward direction but no updations are allowed.
In order to make the non-scrollable ResultSet as scrollable ResultSet we must use the following ***createStatement()*** method which is present in Connection interface.

```
Statement st=conn.createStatement(int Type, int Mode);
```

Here t*ype* represents the type of scrollability and *mode* represents either read only or updatable.
The value of Type and the Modes are present in ResultSet interface as constant data members and they are:

- TYPE_FORWARD_ONLY -> 1
- TYPE_SCROLL_INSENSITIVE -> 2
- CONCUR_READ_ONLY -> 3

For Example,

```
Statement st=con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

Whenever we create a ResultSet object, by default, constant-1 as a Type and constant-3 as mode will be assigned.
ResultSet interface provides us several methods to make an ResultSet as Scrollable ResultSet below is the list of methods available in ResultSet interface.

- **public boolean next ();** It returns true when rs contains next record otherwise false.
- **public void beforeFirst ();** It is used for making the ResultSet object to point to just before the first record (it is by default)
- **public boolean isFirst ();** It returns true when rs is pointing to first record otherwise false.
- **public void first ();** It is used to point the ResultSet object to first record.
- **public boolean isBeforeFirst ();** It returns true when rs pointing to before first record otherwise false.
- **public boolean previous ();** It returns true when rs contains previous record otherwise false.
- **public void afterLast ();** It is used for making the ResultSet object to point to just after the last record.

- **public boolean isLast ();** It returns true when rs is pointing to last record otherwise false.
- **public void last ();** It is used to point the ResultSet object to last record.
- **public boolean isAfterLast ();** It returns true when rs is pointing after last record otherwise false.
- **public void absolute (int);** It is used for moving the ResultSet object to a particular record either in forward direction or in backward direction with respect to first record and last record respectively. If int value is positive, rs move in forward direction to that with respect to first record. If int value is negative, rs move in backward direction to that with respect to last record.
- **public void relative (int);** It is used for moving rs to that record either in forward direction or in backward direction with respect to current record.

```
import java.sql.*;
public class Example1 {
     public static void main(String[] args)
         { try{
              Class.forName("com.mysql.jdbc.Driver");
              String
              dburl="jdbc:mysql://localhost:3306/sixth";
              Connection conn=DriverManager
                          .getConnection(dburl,"root","");

              String sql="SELECT * FROM
              employee"; Statement
              st=conn.createStatement
                         (ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_READ_ONLY);

             ResultSet rs=st.executeQuery(sql);
              //selecting all data
              System.out.println("Records in Student
              Table:");while(rs.next()) {
                    System.out.println(rs.getInt(1)+" "+rs.getString(2));
              }

              //goto first record
              System.out.println("First Record:");
              rs.first();
              System.out.println(rs.getInt(1)+" "+rs.getString(2));

              //goto last record
              System.out.println("Last Record:");
              rs.last();
              System.out.println(rs.getInt(1)+" "+rs.getString(2));

              //goto third record
              System.out.println("Third Record:");
              rs.absolute(3);
```

```
                System.out.println(rs.getInt(1)+" "+rs.getString(2));

                //previous record of third record
                System.out.println("Previous Record:");
                rs.previous();
                System.out.println(rs.getInt(1)+" "+rs.getString(2));

                //using relative function
                System.out.println("Relative Test:");
                rs.relative(-1);
                System.out.println(rs.getInt(1)+" "+rs.getString(2));

            }catch(Exception ex) {
                 System.out.println(ex.toString());
            }
        }
}
```

Output:

```
Records in Student Table:
101 Ram
102 Shyam
103 Hari
104 Gita
First Record:
101 Ram Last
Record:
104 Gita
Third Record:
103 Hari
Previous Record:
102 Shyam
Relative Test:
101 Ram
```

JDBC Updatable ResultSet:

To make the ResultSet object updatable and scrollable we must use the following constants which are present in the ResultSet interface.

- TYPE_SCROLL_SENSITIVE
- CONCUR_UPDATABLE

```
Statement st=con.createStatement (ResultSet.TYPE_SCROLL_SENSITIVE,
                  ResultSet.CONCUR_UPDATABLE);
```

On the above ResultSet object, we can perform the following three operations:

- inserting a record,
- deleting a record and
- updating a record.

Steps to insert a record through ResultSet object:

We can insert the record in the database through ResultSet object using ***absolute()* method**, but before inserting a record, you need to decide at which position you are inserting, since the *absolute()* method takes a position as a parameter where it to be inserted.

Step 1:

```
rs.absolute (3);
```

Step 2:

Since we are inserting a record we must use the following method to make the ResultSet object hold the record.

```
rs.moveToInsertRow ();
```

Step 3:

Update all columns of the database or provide the values to all columns of the database by using the following generalized method which is present in the ResultSet interface.

```
rs.updateXXX(int colno, XXX val);
```

Example:

```
rs.updateInt (1, 105);
rs.updateString (2, “Hari”);
```

Step-4 :

Up to step-3, the data is inserted in the ResultSet object and whose data must be inserted in the database permanently by calling the following method:

```
public void insertRow();
```

By calling the above insertRow() method, the record can be inserted into the database permanently.

Steps to Delete a record through ResultSet :

First, you need to decide which record you need to delete because you need to pass the position of the record to absolute() to point the resultset to a particular record.

```
rs.absolute (3); // rs pointing to 3 rd record & marked for deletion
```

To delete the record permanently from the database we must call the deleteRow() method which is present in ResultSet interface.

```
rs.deleteRow ();
```

Steps for UPDATING a record through ResultSet:

First, you need to decide which record you need to update because you need to pass the position of the record to absolute() to point the resultset to a particular record.

```
rs.absolute (2);
```

And then decide which column to update.

```
rs.updateString (2, “Hari”);
rs.updateInt (1, 105);
```

Using step-2 we can modify the content of the ResultSet object and the content of the ResultSet object must be updated to the database permanently by calling the following method which is present in the ResultSet interface.

```
rs.updateRow ();
```

Example of updatable Result Set

```
import java.sql.*;
public class Example1 {
     public static void main(String[] args)
         { try{
                Class.forName("com.mysql.jdbc.Driver");
                String
                dburl="jdbc:mysql://localhost:3306/sixth";
                Connection conn=DriverManager.
                           getConnection(dburl,"root","");
                String sql="SELECT * FROM employee";
                Statement st=conn.createStatement
                     (ResultSet.TYPE_SCROLL_INSENSITIVE,
                            ResultSet.CONCUR_UPDATABLE);
                ResultSet rs=st.executeQuery(sql);
                //selecting all data
                System.out.println("Records in Student Table:");
                while(rs.next()) {
                     System.out.println(rs.getInt(1)+" "+rs.getString(2));
                 }

                //updating second row
                rs.absolute(2);
                rs.updateString(2, "Ram Sharma");
                rs.updateRow();
                System.out.println("1 row updated!");

                //inserting a row
                rs.moveToInsertRow();
                rs.updateInt(1, 103);
                rs.updateString(2, "Hari Sharma");
                rs.insertRow();
                System.out.println("1 row inserted!");
                //selecting all data again
                System.out.println("Records in Student Table:");
                rs.beforeFirst();
                while(rs.next()) {
                      System.out.println(rs.getInt(1)+" "+rs.getString(2));
                }

                conn.close();

          }catch(Exception ex) {
                System.out.println(ex.toString());
          }
      }
}
```

JDBC RowSet

A JDBC RowSet facilitates a mechanism to keep the data in tabular form. It happens to make the data more flexible as well as easier as compared to a ResultSet. The connection between the data source and the RowSet object is maintained throughout its life cycle. The RowSet supports development models that are component-based such as JavaBeans, with the standard set of properties and the mechanism of event notification.

The implementation classes of the RowSet interface are as follows:

- JdbcRowSet
- CachedRowSe
t ○ WebRowSet ○
JoinRowSet
- FilteredRowSet

Creating and Executing a RowSet:

```
 JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
rowSet.setUrl("jdbc:mysql://localhost:3306/dbname");
rowSet.setUsername("root");
rowSet.setPassword("");

 rowSet.setCommand("select * from
employee"); rowSet.execute();
```

The advantages of using RowSet are given below:

1. It is easy and flexible to use.
2. It is Scrollable and Updatable by default.

Example of using a RowSet

```
import java.sql.*;
import javax.sql.rowset.*;
```

```
public class Example1 {
     public static void main(String[] args)
         { try{
             Class.forName("com.mysql.jdbc.Driver");

             JdbcRowSet jrs=RowSetProvider.
                     newFactory().createJdbcRowSet();
             jrs.setUrl("jdbc:mysql://localhost:3306/sixth");

             jrs.setUsername("root"); jrs.setPassword("");

             String sql="SELECT * FROM
             employee"; jrs.setCommand(sql);
             jrs.execute();
              //selecting record
             while(jrs.next()) {
                   System.out.println(jrs.getInt(1)+" "
                         +jrs.getString(2));
             }

         }catch(Exception ex) {
               System.out.println(ex.toString());
         }
     }
}
```

Output:

```
101 Ram
102 Ram Sharma
103 Hari Sharma
```

Cached RowSet

A cached row set is a row set implementation where the rows are cached and the row set does not have a live connection to the database (disconnected). It is also called disconnected Row Set.

Example

```
CachedRowSet crs=RowSetProvider.newFactory().createCachedRowSet();
crs.setUrl("jdbc:mysql://localhost:3306/sixth");
crs.setUsername("root");
crs.setPassword("");

String sql="SELECT * FROM
employee"; crs.setCommand(sql);
crs.execute();
```

Batch Processing in JDBC

Batch processing is a process of keeping multiple queries on a batch and executing all the queries on a batch at once. In Batch Processing, the SQL operations will be constructed as a batch and then the batch will be send to database in a single trip.

To perform the batch processing in JDBC, the Statement interface provided two methods.

- addBatch()
- executeBatch()

addBatch() :

addBatch() method is used to construct a batch. Constructing a batch means, storing the SQL commands in a buffer, maintained by Statement object.

executeBatch() :

To execute the batch, we will use the executeBacth() method. When **executeBatch()** called, then the commands will be transferred at a time as a batch from buffer to database.

```
import java.sql.*;
public class Example1 {
     public static void main(String[] args)
         { try{
             Class.forName("com.mysql.jdbc.Driver");
             String
             dburl="jdbc:mysql://localhost:3306/sixth";
             Connection conn=DriverManager.
                     getConnection(dburl,"root","");

             Statement st=conn.createStatement();

             st.addBatch("insert into employee(eid,name)
                          values (101,'Ram')");
             st.addBatch("insert into employee(eid,name)
                        values (102,'Hari')");
             st.addBatch("update employee set
                       name='Guru' WHERE eid=101");

             st.executeBatch();
             System.out.println("Batch Executed Successfully!");

             //selecting records
             String sql="select * from employee";
             ResultSet rs=st.executeQuery(sql);
             while(rs.next()) {
                  System.out.println(rs.getInt(1)+" "+rs.getString(2));
             }

             st.close();
```

```
                conn.close();

          }catch(Exception ex) {
                System.out.println(ex.toString())
                ;
          }
      }
 }
```

Output:

```
Batch Executed Successfully!
101 Guru
102 Hari
```

Transaction Management in JDBC

- A transaction means, it is a group of operations used to perform a task.
- A transaction can reach either success state or failure state.
- If all operations are completed successfully then the transaction becomes success.
- If any one of the operation fail then all remaining operations will be cancelled and finally transaction will reach to fail state.

Types of Transactions :

The basic transactions are of two types.

- Local Transactions
- Global / Distributed Transactions

Local Transactions :

If all the operations are executed on one/same database, then it is called as local transaction.

Global / Distributed Transaction :

If the operations are executed on more than one database then it is called as global transactions.

We can get the Transaction support in JDBC from Connection interface. The Connection interface given 3 methods to perform Transaction Management in JDBC.

- **setAutoCommit()**
- **commit()**
- **rollback()**

Transaction setAutoCommit() :

Before going to begin the operations, first we need to disable the auto commit mode. This can be done by calling setAutoCommit(false).

By default, all operations done from the java program are going to execute permanently in database. Once the permanent execution happened in database, we can't revert back them (Transaction Management is not possible).

Transaction commit() :

If all operations are executed successfully, then we commit a transaction manually by calling the **commit()** method.

Transaction rollback() :

If any one of the operation failed, then we cancel the transaction by calling **rollback**() method.

```
connection.setAutoCommit(false);

try{

----------
----------

    connection.commit();

}catch(Exception e){

    connection.rollback();

}
```

Example is shown below:

```
import java.sql.*;
public class Example1 {
     public static void main(String[] args) throws SQLException
           { Connection conn=null;
           String sql="";
           Statement st=null;
         try{
             Class.forName("com.mysql.jdbc.Driver");
             String dburl="jdbc:mysql://localhost:3306/sixth";
             conn=DriverManager.getConnection(dburl,"root","");

             //setting auto commit false
             conn.setAutoCommit(false);

             st=conn.createStatement();
             sql="insert into employee(eid,name) values
             (101,'Ram')"; st.execute(sql);
             //commit this insert - will not be rolled
             back conn.commit();
             System.out.println("Transaction Committed!");

             //another insert without commit
             sql="insert into employee(eid,name)
             values (102,'Hari')"; st.execute(sql);
```

```
              //another insert with exception
              sql="insert into employee(eid1,name1)
                     values (103,'Gita')";
              st.execute(sql);

              st.close();
              conn.close();
         }catch(Exception ex) {
               //if failed we can rollback
               conn.rollback();
               System.out.println("Transaction Rolled Back!");

               //selecting data after rolled back
               ResultSet rs=st.executeQuery("select * from employee");
               while(rs.next()) {
                     System.out.println(rs.getString(1)+" "+
                            rs.getString(2));
               }
         }
      }
 }
```

Output:

```
Transaction Committed!
Transaction Rolled Back!
101 Ram
```