1. Database Connectivity (JDBC):

A database is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation. A database management system (DBMS) provides mechanisms for storing, organizing, retrieving and modifying data from any users. DBMS allows for the access and storage of data without concern for the internal representation of data.

Java Database Connectivity (JDBC) is an API (Application Programming Interface) for java that allows the Java programmer to access the database. The JDBC API consists of a numbers of classes and interfaces, written in java programming language, which provides a numbers of methods for updating and querying a data in a database. It is a relational database oriented driver. It makes a bridge between java application and database. Java application utilizes the features of database using JDBC.

JDBC helps to write Java applications that manage these three programming activities:

- Connect to a data source, like a database.
- Send queries and update statements to the database.
- Retrieve and process the results received from the database in answer to our query.

ODBC (Open DataBase Connectivity)

ODBC is a standard database access method developed by the SQL Access group. The goal of ODBC is to make it possible to access any data from any application, regardless of which database management system (DBMS) is handling the data. ODBC manages this by inserting a middle layer, called a database driver, between an application and the DBMS. The purpose of this layer is to translate the application's data queries into commands that the DBMS understands. For this to work, both the application and the DBMS must be ODBC-compliant i.e. the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.

1.1 JDBC Components:

- **The JDBC API:**

  The JDBC API provides programmatic access to relational data from the Java programming language. Using JDBC API, front end java applications can execute query and fetch data from connected database. JDBC API can also connect with multiple applications with same database or same application with multiple databases which can resides in different computers (distributed environment). The JDBC API is part of the Java platform, which includes the Java Standard Edition (Java SE) and the Java Enterprise Edition (Java EE). The JDBC 4.0 API is divided into two packages: java.sql and javax.sql. Both packages are included in the Java SE and Java EE platforms.

- **JDBC Driver Manager**:

  The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. Driver Manager manages all the JDBC Driver loaded in client's system. Driver Manager loads the most appropriate driver among all the Drivers for creating a connection.

- **JDBC Test Suite**:

  The JDBC driver test suite helps us to determine that JDBC drivers will run our program. JDBC Test Suite is used to check compatibility of a JDBC driver with J2EE platform. It also check whether a Driver follow all the standard and requirements of J2EE Environment.

- **JDBC-ODBC Bridge:**

  The Java Software Bridge provides JDBC access via ODBC Bridge (or drivers). JDBC Bridge is used to access ODBC drivers installed on each client machine. The JDBC Driver contacts the ODBC Driver for connecting to the database. When Java first came out, ODBC was a useful driver because

most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available. The ODBC Driver is already installed or come as default driver in windows. In Windows 'Datasource' name can be created using control panel >administrative tools>Data Sources (ODBC). After creating 'data source', connectivity of 'data source' to the 'database' can be checked. Using this 'data source' , we can connect JDBC to ODBC.

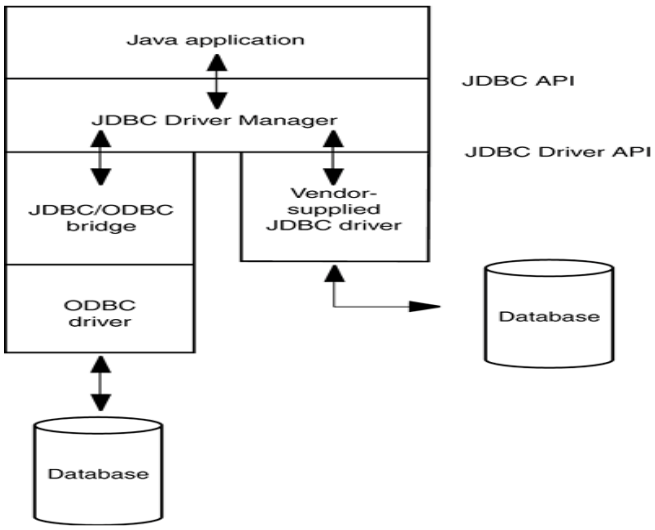


Fig: JDBC-to-database communication path

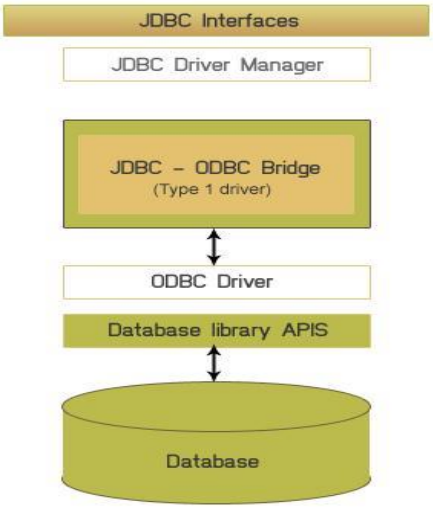
1.2 JDBC Driver Types

JDBC drivers are classified into the following four categories:

❖ JDBC-ODBC BRIDGE DRIVER(TYPE 1)

Features

- Convert the query of JDBC Driver into the ODBC query, which in return pass the data.
- The bridge requires deployment and proper configuration of an ODBC driver.
- JDBC-ODBC is native code not written in java.
- Nowadays in most cases it is only being used for the educational purposes.
- The connection occurs as follows -- Client -> JDBC Driver -> ODBC Driver -> Database.



Pros

A type-1 driver is easy to install and handle.

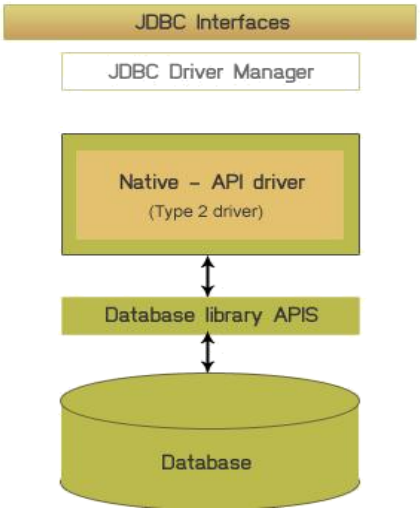
Cons

- Extra channels in between database and application made performance overhead.
- Needs to be installed on client machine.
- Not suitable for applet, due to the installation at clients end.

❖ Native-API (Type-2) Driver

Features

The type 2 driver need libraries installed at client site. For example, we need "mysqlconnector.jar" to be copied in library of java kit. It is not written in java entirely because the non-java interfaces have the direct access to database. It is written partly in Java and partly in native code. When we use such a driver, we must install some platform-specific code in addition to a Java library. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.



Pros

- Type 2 driver has additional functionality and better performance than Type 1.
- Has faster performance than type 1,3 and 4,since it has separate code for native APIS.

Cons

- Library needs to be installed on the client machine.
- Due to the client side software demand, it can't be used for web based application.
- Platform dependent.
- It doesn't support "Applets".

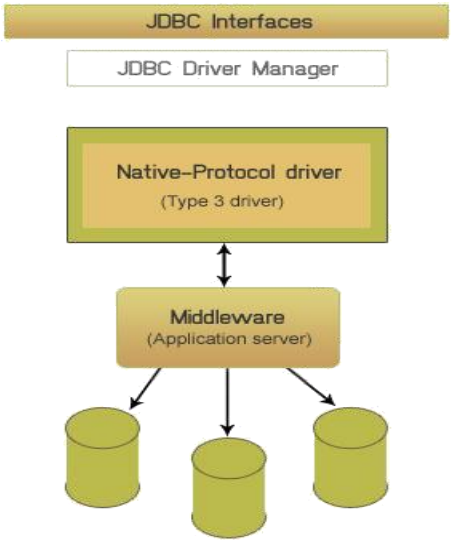
❖ Network-Protocol (Type 3) driver

Features

- It is also known as JDBC-Net pure Java.
- It has 3-tier architecture.
- It can interact with multiple database of different environment.
- The JDBC Client driver written in java communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database

commands for that database.

- The connection occurs as follows--Client -> JDBC Driver -> Middleware-Net Server -> Any Database.



Pros

- The client driver to middleware communication is database independent.
- Can be used in internet since there is no client side software needed.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

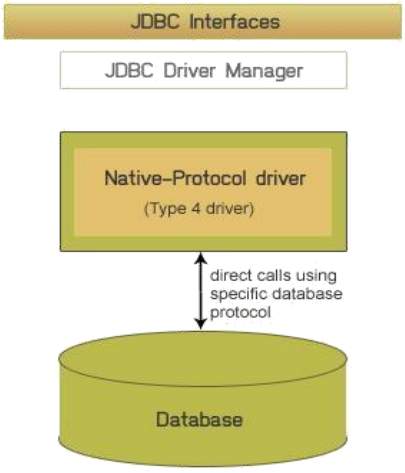
Cons

- Needs specific coding for different database at middleware.
- Due to extra layer in middle can result in time-delay.

❖ Native Protocol (Type 4) Driver

Features

Also known as Direct to Database Pure Java Driver .It is entirely in java. It interacts directly with database generally through socket connection. It is platform independent. It directly converts driver calls into database protocol calls. MySQL's Connector/J driver is an example of Type 4 driver.



Pros

- Improved performance because no intermediate translator like JDBC or middleware server.
- All the connection is managed by JVM, so debugging is easier.

2. Creating JDBC Application:

There are following six steps involved in building a JDBC application:

- **Import the packages:** We should include the packages containing the JDBC classes needed for database programming. We use,

  import java.sql.*.
- **Register (Load) the JDBC driver:** We should initialize a driver so that we can open a communications channel with the database. We can load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself.

  Syntax: Class.forName (String ClassName)
  Example:Class.forName(”sun.jdbc.odbc.JdbcOdbcDriver”);
- **Creating a jdbc Connection:** We can use the DriverManager.getConnection() method to create a Connection object, which represents a physical connection with the database. The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager class manages the JDBC drivers that are installed on the system and getConnection() method is used to establish a connection to a database. This method uses a jdbc url and username, password (not compulsory in every DBMS) to establish a connection to the database and returns a connection object.

  Syntax: Connection con=DriverManager.getConnection(url);
- **Creating a jdbc Statement object:** Once a connection is obtained we can interact with the database. To execute SQL statements, we need to instantiate a Statement object from our connection object by using the createStatement() method.
  Syntax: Statement statement = con.createStatement();
  A statement object is used to send and execute SQL statements to a database.
- **Executing a SQL statement with the Statement object, and returning a jdbc resultSet:** Object of type Statement is used for building and submitting an SQL statement to the database. The Statement class has three methods for executing statements i.e. executeQuery(), executeUpdate(), and execute(). For a SELECT statement, executeQuery() method is used . For statements that create or modify tables, executeUpdate() method is used. execute() executes an SQL statement that is written as String object.

  ResultSet provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.
- **Clean up the environment:** We should explicitly close all database resources after the task is complete.

Sample Code:

```
//STEP 1. Import required packages
import java.sql.*;
public class FirstExample {
   static final String JDBC_DRIVER = " sun.jdbc.odbc.JdbcOdbcDriver ";
   static final String DB_URL = "jdbc:odbc:EMP";

   public static void main(String[] args) {
   Connection conn = null;
   Statement stmt = null;
   try{
      //STEP 2: Register JDBC driver
      Class.forName(JDBC_DRIVER);
      //STEP 3: Open a connection
      System.out.println("Connecting to database...");
      conn = DriverManager.getConnection(DB_URL);
      //STEP 4: create a statement object
      System.out.println("Creating statement...");
      stmt = conn.createStatement();
     //STEP 5:Execute a SQL statement with the Statement object, and returning a jdbc resultSet
      String sql = "SELECT id, first, last, age FROM Employees";
      ResultSet rs = stmt.executeQuery(sql);
      //Extract data from result set
      while(rs.next()){
         //Retrieve by column name
         int id  = rs.getInt("id");
         int age = rs.getInt("age");
         String first = rs.getString("first");
         String last = rs.getString("last");
         //Display values
         System.out.print("ID: " + id);
         System.out.print(", Age: " + age);
         System.out.print(", First: " + first);
         System.out.println(", Last: " + last);
      }
      //STEP 6: Clean-up environment
      rs.close();
      stmt.close();
      conn.close();
   }catch(SQLException se){
         JOptionPane.showMessageDialog(null, “SQL exception caught”);
   }catch(Exception e){
         JOptionPane.showMessageDialog(null, “exception caught”);
   }
   }//end main
}//end FirstExample
```

OUTPUT:

```
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Dipendra, Last: Pandey
ID: 101, Age: 25, First: Madan, Last: Pandey
ID: 102, Age: 30, First: Ram, Last: Bhatt
ID: 103, Age: 28, First: Hari, Last: Pant
```

3. Types of Statement Objects

Generally there are three types of statement objects. They are: Statement, PreparedStatement and CallableStatement. These statement interfaces define the methods and properties that enable us to send SQL or PL/SQL commands and receive data from our database. They also define methods that help bridge data type differences between Java and SQL data types used in a database.

Generally, three types of parameters exist: IN, OUT, and INOUT.

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. We bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. We retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. We bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

3.1 The Statement:

The Statement object is used for general-purpose access to our database. It represents the base statements interface. It is suitable to use the Statement only when we know that we will not need to execute the SQL query multiple times. The Statement interface cannot accept parameters. In contrast to PreparedStatment, the Statement does not offer support for the parameterized SQL queries. Before we can use a Statement object to execute a SQL statement, we need to create the object using the Connection object's createStatement( ) method. Statements would be suitable for the execution of the DDL statements such as CREATE, ALTER, DROP.

3.2 PreparedStatements

The PreparedStatement is derived from the more general class, Statement. The PreparedStatement interface extends the Statement interface which gives us added functionality over a generic Statement object. This statement gives us the flexibility of supplying arguments dynamically. It is more efficient to use the PreparedStatement because the SQL statement that is sent gets pre-compiled in the DBMS. We can also use PreparedStatement to safely provide values to the SQL parameters, through a range of setter methods (i.e. setInt (int, int), setString (int, String), etc.). We can execute the same query repeatedly with different parameter values. The PreparedStatement interface accepts input parameters at runtime i.e. the PreparedStatement object only uses the IN parameter. Just as we close a Statement object, we should also close the PreparedStatement object.

Example:

```
try {
        connection = DriverManager.getConnection(
          DATABASE_URL);
   PreparedStatement pstmt = connection.prepareStatement(
                        "INSERT INTO authors VALUES(?, ?, ?)");
pstmt.setInt(1,19);
pstmt.setString(2, "Navin");
pstmt.setString(3,"Sharma");

pstmt.execute();

}catch ( SQLException e )  {
       e.printStackTrace();
     } // end catch
```

The three question marks (?) in the the preceding SQL statement's last line are placeholders for values that will be passed as part of the query to the database. Before executing a PreparedStatement, the program must specify the parameter values by using the Prepared- Statement interface's set methods.
For the preceding query, parameters are int and strings that can be set with Prepared- Statement method setInt and setString.
Method setInt's and setString's first argument represents the parameter number being set, and the second argument is that parameter's value. Parameter numbers are counted from 1, starting with the first question mark (?).
Interface PreparedStatement provides set methods for each supported SQL type. It's important to use the set method that is appropriate for the parameter's SQL type in the database—SQLExceptions occur when a program attempts to convert a parameter value to an incorrect type.

3.3 CallableStatement

The CallableStatement extends PreparedStatement interface. It is used when we want to access database stored procedures. The main advantage of this statement is that it adds a level of abstraction, so the execution of stored procedures does not have to be DBMS-specific. The CallableStatement interface can also accept runtime input parameters. The CallableStatement object can use all three parameters IN, OUT, and INOUT. The output parameters needs to be explicitly defined through the corresponding registerOutParameter() method, whereas the input parameters are provided in the same manner as with the PreparedStatement. Just as we close other Statement object, we should also close the CallableStatement object.

4. Types of ResultSet, ResultSetMetadata

4.1 ResultSet

A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. A ResultSet object can be created through any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet.

We access the data in a ResultSet object through a cursor. This cursor is a pointer that points to one row of data in the ResultSet. Initially, the cursor is positioned before the first row. The method **ResultSet.next()** moves the cursor to the next row. This method returns false if the cursor is positioned after the last row. This method repeatedly calls the ResultSet.next() method with a while loop to iterate through all the data in the ResultSet.

ResultSet Interface

The ResultSet interface provides methods for retrieving and manipulating the results of executed queries, and ResultSet objects can have different functionality and characteristics. These characteristics are **type, concurrency, and cursor holdability.**

4.2 Types of ResultSet

The type of a ResultSet object determines the level of its functionality in two areas: the ways in which the cursor can be manipulated, and how concurrent changes made to the underlying data source are reflected by the ResultSet object. There are generally three different ResultSet types:

- **TYPE_FORWARD_ONLY:** The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- **TYPE_SCROLL_INSENSITIVE:** The result set can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- **TYPE_SCROLL_SENSITIVE:** The result set can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

The default ResultSet type is TYPE_FORWARD_ONLY.

Note: Not all databases and JDBC drivers support all ResultSet types. The method DatabaseMetaData.supportsResultSetType returns true if the specified ResultSet type is supported and false otherwise.

4.3 ResultSetMetadata

The metadata describes the ResultSet's contents. Programs can use metadata programmatically to obtain information about the ResultSet's column names and types. ResultSetMetaData method getColumnCount is used to retrieve the number of columns in the ResultSet.

5. CRUD Operations In Database:

The operations such as insertion, creation, deletion, updating and selection of the tabular data are known as CRUD operation in database. Following are the CRUD operations with the appropriate examples;

INSERT OPERATION:

Example1: Program that insert data in the student table through java

```
import java.sql.*;
class TestJdbc
{
public static void main(String arg[])
{
        try{
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                Connection cn = DriverManager.getConnection("jdbc:odbc:ram");
                 String str = "insert into student values(12,'Ram', 'dhangadi')";
                Statement stat = cn.createStatement();
                stat.executeUpdate(str);
                cn.close();
        }
        catch(ClassNotFoundException e)
        {
                System.out.println("Cannot insert into the table");
        }
        catch(SQLException e) {}
}
}
```

Example2: Java program that is used to insert multiple records in a table Customers

```
import java.sql.*;
class JdbcInsertMultipleRows {
  public static void main (String[] args) { try
    {
      String url = "jdbc:odbc:ram";
      Connection conn = DriverManager.getConnection(url,"","");
      Statement st = conn.createStatement();
      st.executeUpdate("INSERT INTO Customers " + "VALUES (1001, 'Ram','Kathmandu', 2001)");
      st.executeUpdate("INSERT INTO Customers " + "VALUES (1002,'Shyam','Pokhara', 2004)");
      st.executeUpdate("INSERT INTO Customers " + "VALUES (1003, 'Hari','Nepalgung', 2003)");
      st.executeUpdate("INSERT INTO Customers " + "VALUES(1004,'Krishna','Dhangadhi',2001)");
      conn.close();
    }
      catch (SQLException e)
      {
      System.out.println("Cannot insert into the table");
       }

  }
```

```
}
```

SELECT OPERATION

Example: Program that is used to select multiple rows from the table of SQL database

```
import java.sql.*;
class Query {
   public static void main (String[] args) { try
      {
         String url = "jdbc:odbc:ram";
         Connection conn = DriverManager.getConnection(url,"","");
         Statement stmt = conn.createStatement();
         ResultSet rs;
         rs = stmt.executeQuery("SELECT name FROM student WHERE id = 1001");
         while ( rs.next() ) {
            String lastName = rs.getString("name");
                  System.out.println(lastName);
         }
         conn.close();
      } catch (SQLException e) {
      System.out.println("Cannot select from the table");

      }
   }
}
```

UPDATE OPERATION:

Example: Program that is used to modify the record in the table of SQL database

```
import java.sql.*;
 class UpdateQuery {
   public static void main (String[] args) { try {
                  String url = "jdbc:odbc:ram";
                  Connection conn = DriverManager.getConnection(url,"","");
                   Statement stmt = conn.createStatement();
                  ResultSet rs;
                  stmt.executeUpdate("UPDATE student SET name = 'Krishna' WHERE id = 1001");
                  rs = stmt.executeQuery("SELECT name from student where id = 1001");
                  while ( rs.next() )
                  {
                           String lastName = rs.getString("name");
                           System.out.println(lastName);
                  }
                  conn.close();
                           } catch (SQLException e) {
                           System.out.println("Cannot update the table");
```

```
        }
        }
        }
```

DELETE OPERATION

Example: Program that is used to delete record from the table of SQL database

```
import java.sql.*;
class DeleteQuery {
   public static void main (String[] args) {
    try {
                String url = "jdbc:odbc:ram";
                Connection conn = DriverManager.getConnection(url,"","");
                 Statement stmt = conn.createStatement();
                ResultSet rs;
                stmt.executeUpdate("DELETE from student WHERE id = 1001");
                conn.close();
        } catch (SQLException e) {
                        System.out.println("Cannot delete the data");

     }
   }
}
```

CREATE OPERATION

Example: Program that is used to create table in the given database

```
import java.sql.*;
class CreateTableQuery {
   public static void main (String[] args) {
    try {
                String url = "jdbc:odbc:ram";
                Connection conn = DriverManager.getConnection(url,"","");
                 Statement stmt = conn.createStatement();
                String sql = "create table college(cname varchar(255), address varchar(255))";
                stmt.executeUpdate(sql);
                System.out.println("Table is created in the given database");
                conn.close();
     } catch (SQLException e) {
                        System.out.println("Cannot create the table");

     }
   }
}
```

6. JDBC and AWT

AWT is responsible for creating the GUI components and JDBC is responsible for database connectivity. java.awt package has all the classes that helps in creation of the GUI components.

```
    //Example for AWT, JDBC
import java.sql.*;
import java.awt.*;
import java.awt.event.*;

class AwtJdbcTest extends Frame implements ActionListener {
   Connection con;
   PreparedStatement ps;
   ResultSet rs;
   Label lno, lna, ladd;
   TextField tno, tna, tadd;
   Button bins;
   AwtJdbcTest() {
      FlowLayout fl;
      fl = new FlowLayout();
      setLayout(fl);
      setSize(500, 400);
      setBackground(Color.GREEN);
      lno = new Label("RollNO");
      add(lno);
      tno = new TextField(20);
      add(tno);
      lna = new Label("Name");
      add(lna);
      tna = new TextField(20);
      add(tna);
      ladd = new Label("Address");
      add(ladd);
      tadd = new TextField(20);
      add(tadd);
      bins = new Button("INSERT");
      bins.addActionListener(this);
      add(bins);
      setVisible(true);
      dbConnect();
          addWindowListener(new WindowAdapter(){
                        public void windowClosing(WindowEvent e)
                         {
                           System.exit(0);
```

```
                        }});

    } // constructor closing
   void dbConnect() {
      try{
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         con = DriverManager.getConnection("jdbc:odbc:school”);
         String q = "INSERT INTO student VALUES(?, ?, ?)";
         ps = con.prepareStatement(q);
        } // try
      catch(Exception e) {
         e.printStackTrace();
      } //catch
   } // dbConnect() closing

   public void actionPerformed(ActionEvent ae) {
      try{
         int no;
         String name, add;
         no = Integer.parseInt(tno.getText());
         name = tna.getText();
         add = tadd.getText();
            ps.setInt(1, no);
             ps.setString(2, name);
             ps.setString(3, add);
             ps.executeUpdate();
          JOptionPane.showMessageDialog(null,"data inserted successfully");
      } // try
      catch(Exception e)    {
         e.printStackTrace();
      } //catch
   } // actionPerformed
   public static void main(String[] args) {
      new AwtJdbcTest();
   } // main
} // class
```

7. Connection Pooling

Object pooling is used as a technique to improve application performance. Object pooling is effective for two simple reasons. First, the run time creation of new software objects is often more expensive in terms of performance and memory than the reuse of previously created objects. Second, garbage collection is an expensive process and when we reduce the number of objects to clean up, we generally reduce the garbage collection load. Objects which do not have short lifetimes are used for pooling. Pooling is the concept that contains various pooled objects such as database connections, process threads, server sockets or any other kind of object that may be expensive to create from scratch. When we create any application, it first asks the pool for objects. If there are no objects in the pool, then they will be newly created otherwise they are obtained from the pool. And when the application has finished with the object it is returned to the pool rather than destroyed. So the pooled object can be reused in the future and hence are less expensive in contrast to create it from scratch.

Database connections are often expensive to create because of the overhead of establishing a network connection and initializing a database connection. JDBC connection pooling is conceptually similar to any other form of object pooling. It setup the pool of database connection objects and instead of creating the database object many times, it simply use the object from the pool when the database object is not being used. JDBC connections are both expensive to initially create and then maintain over time. Therefore, they are an ideal resource to pool.