

## Unit-3 JDBC

- 3.1 Client-Server Design: Two-Tier Database Design, Three-Tier Database Design
- 3.2 The JDBC API: The API Components, Database Creation, table creation using SQL
- 3.3 JDBC Database Example
- 3.4 JDBC Drivers
- 3.5 JDBC-ODBC Bridge
- 3.6 JDBC-Advantages & Disadvantages

### JDBC:

- **JDBC** stands for **Java Database Connectivity**.
- JDBC was developed by JavaSoft of Sun Microsystems.
- **JDBC** is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases
- It defines interfaces and classes for writing database applications in Java by making database connections. Using JDBC you can send SQL, PL/SQL statements to almost any relational database.
- JDBC is a Java API for executing SQL statements and supports basic SQL functionality. It provides RDBMS access by allowing you to embed SQL inside Java code. Since nearly all relational database management systems (RDBMSs) support SQL, and because Java itself runs on most platforms, JDBC makes it possible to write a single database application that can run on different platforms and interact with different DBMSs.
- Java Database Connectivity is similar to Open Database Connectivity (ODBC) which is used for accessing and managing database, but the difference is that JDBC is designed specifically for Java programs, whereas ODBC is not depended upon any language.

In short JDBC helps the programmers to write java applications that manage these three programming activities:

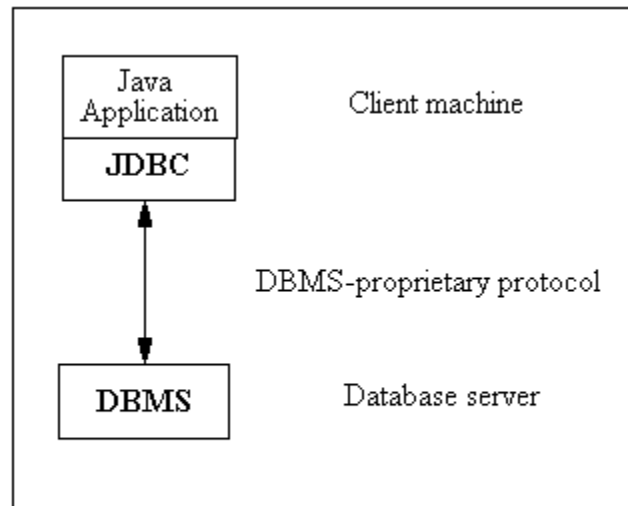
- Connect to a data source, like a database.
- Send queries and update statements to the database.
- Retrieve and process the results received from the database in answer to your query.

### ➤ JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access.

#### 1. Two-tier Architecture for Data Access or Two-tier database design

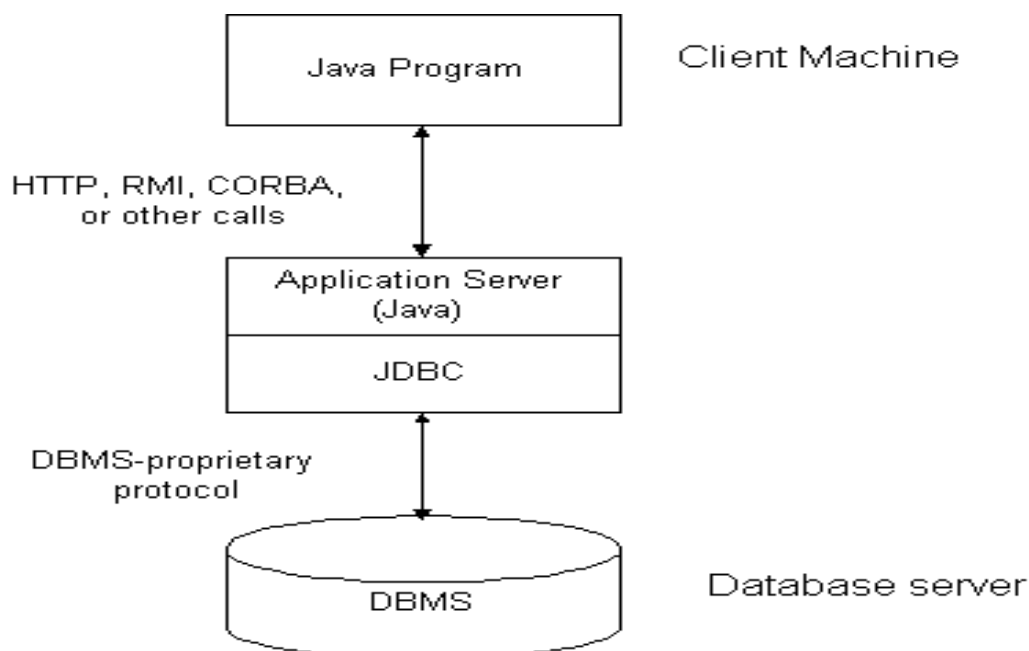
- In the two-tier model, a **Java application talks directly to the data source**.



Two-tier model

- This requires a JDBC driver that can communicate with the particular data source being accessed.
- A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user.
- The data source may be located on another machine to which the user is connected via a network. This is referred to as a **client/server configuration**, with the user's machine as the **client**, and the machine housing the data source as the **server**.
- The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

### Three-tier Architecture for Data Access



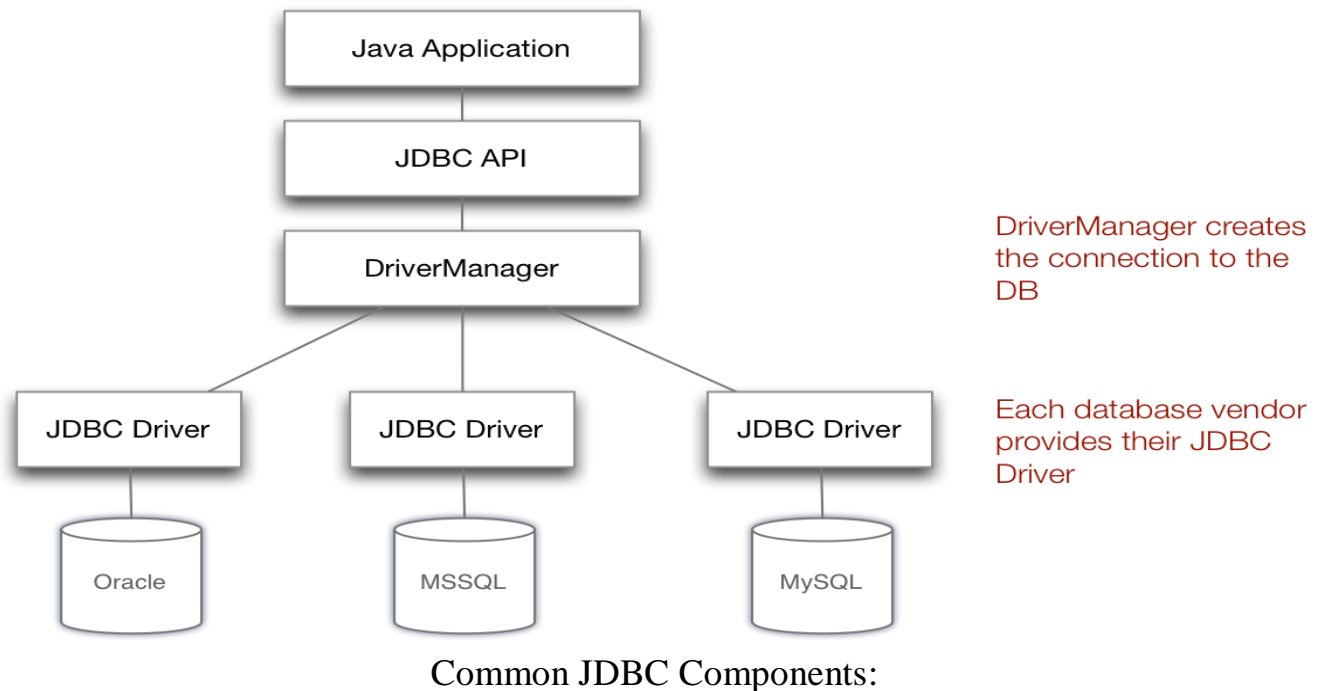
- In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source.
- The data source processes the commands and sends the results back to the middle tier, which then sends them to the user.
- MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data.
- Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

### Figure 2: Three-tier Architecture for Data Access.

- Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java byte code into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.
- With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

### General JDBC Architecture

- The JDBC API supports both two-tier and three-tier processing models for database access but in **general JDBC Architecture** consists of two layers:
  - **JDBC API:** This provides the application-to-JDBC Manager connection.
  - **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.
- The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.
- The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.
- Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application:



### ❖ JDBC Product Components:

JDBC has four Components:

1. The JDBC API.
2. The JDBC Driver Manager.
3. The JDBC Test Suite.
4. The JDBC-ODBC Bridge.

#### 1. JDBC API:

- The JDBC API provides the facility for accessing the relational database from the Java programming language.
- The API technology provides the industrial standard for independently connecting Java programming language and a wide range of databases. The user not only execute the SQL statements, retrieve results, and update the data but can also access it anywhere within a network because of its "Write Once, Run Anywhere" (WORA) capabilities.
- Due to JDBC API technology, user can also access other tabular data sources like spreadsheets or flat files even in the heterogeneous environment.
- JDBC application programming interface is a part of the Java platform that have included in Java Standard Edition (Java SE) and the Java Enterprise Edition (Java EE) in itself.
- The latest version of JDBC 4.0 application programming interface is divided into two packages
  - i. **java.sql**
  - ii. **javax.sql**.

#### 2. JDBC Driver Manager:

- This interface manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub

protocol.

- The first driver that recognizes a certain sub protocol under JDBC will be used to establish a database Connection.
- Internally, JDBC DriverManager is a class in JDBC API. The objects of this class can connect Java applications to a JDBC driver.
- DriverManager is the very important part of the JDBC architecture. The main responsibility of JDBC DriverManager is to load all the drivers found in the system properly.
- The Driver Manager also helps to select the most appropriate driver from the previously loaded drivers when a new open database is connected.

### 3: JDBC Test Suite:

- The function of JDBC driver test suite is to make ensure that the JDBC drivers will run user's program or not.
- The test suite of JDBC application program interface is very useful for testing a driver based on JDBC technology during testing period. It ensures the requirement of Java Platform Enterprise Edition (J2EE).

### 4. JDBC-ODBC Bridge:

- The JDBC-ODBC Bridge, also known as JDBC type 1 driver is a database driver that utilizes the ODBC driver to connect the database.
- This driver translates JDBC method calls into ODBC function calls. The Bridge implements JDBC for any database for which an ODBC driver is available. The Bridge is always implemented as the **sun.jdbc.odbc**
- Java package and it contains a native library used to access ODBC.

## JDBC API

The JDBC API provides the following interfaces and classes:

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain sub protocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use a DriverManager object, which manages objects of this type. It also abstracts the details associated with working with Driver objects
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.

- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

## Important classes and interfaces

The mechanism to communicate with database server exists as standard java classes.

- JDBC API has abstract java interfaces that allow to link to any number of databases.
- **java.sql.DriverManager:** handles the loading and unloading of appropriate database drivers required to make the connection.
- **java.sql.Connection:** exposes the database to the developer ,and give connection as an java component.
- **java.sql.Statement:** provides a container for executing SQL statement using connection.
- **java.sql.ResultSet:** represent the data that returns form the database server to java application.

### 1) Driver manager class

- The **DriverManager** class acts as an interface between user and drivers.
- It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.
- The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method **DriverManager.registerDriver()**.

Commonly used **methods** of DriverManager class:

1)	public static void registerDriver(Driver driver):	is used to register the given driver with DriverManager.
2)	public static void deregisterDriver(Driver driver):	is used to deregister the given driver (drop the driver from the list) with DriverManager.
3)	public static Connection <b>getConnection</b> (String url):	is used to establish the connection with the specified url.
4)	<b>getConnection</b> (String url,String userName,String password):	is used to establish the connection with the specified url, username and password.
5)	<b>getDrivers()</b>	Access a list of drivers present in database
6)	<b>println( )</b>	Prints a message used in log stream

## 2) Connection interface:

- A Connection is the session between java application and database.
- The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData.
- The Connection interface provide many methods for transaction management like commit(),rollback() etc
- By default, connection commits the changes after executing queries.

public Statement <b>createStatement</b> ():	creates a statement object that can be used to execute SQL queries.
public Statement <b>createStatement</b> (int resultSetType,int resultSetConcurrency):	Creates a Statement object that will generate ResultSet objects with the given type and concurrency.
public void setAutoCommit(boolean status):	is used to set the commit status.By default it is true.
public void commit():	saves the changes made since the previous commit/rollback permanent.
public void rollback():	Drops all changes made since the previous commit/rollback.
public void <b>close</b> ():	closes the connection and Releases a JDBC resources immediately.

## 3) Statement interface

- The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

public ResultSet <b>executeQuery</b> (String sql):	is used to execute SELECT query. It returns the object of ResultSet.
public int <b>executeUpdate</b> (String sql):	is used to execute specified query, it may be create, drop, insert, update, delete etc.
public boolean <b>execute</b> (String sql):	is used to execute queries that may return multiple results.
public int[] <b>executeBatch</b> ():	is used to execute batch of commands.



#### 4) **ResultSet** interface

- The object of ResultSet maintains a cursor pointing to a particular row of data. Initially, cursor points to before the first row.
- It is used to fetch record from database for displaying in output.
- By default, ResultSet object can be moved forward only and it is not updatable

Commonly used methods of ResultSet interface

public boolean <b>next()</b> :	is used to move the cursor to the one row next from the current position.
public boolean <b>previous()</b> :	is used to move the cursor to the one row previous from the current position.
public boolean <b>first()</b> :	is used to move the cursor to the first row in result set object.
public boolean <b>last()</b> :	is used to move the cursor to the last row in result set object.
public boolean <b>absolute</b> (int row):	is used to move the cursor to the specified row number in the ResultSet object.
public boolean <b>relative</b> (int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
public int <b>getInt</b> (int columnIndex):	is used to return the data of specified column index of the current row as int.
public int <b>getInt</b> (String columnName):	is used to return the data of specified column name of the current row as int.
public String <b>getString</b> (int columnIndex):	is used to return the data of specified column index of the current row as String.
public String <b>getString</b> (String columnName):	is used to return the data of specified column name of the current row as String.

#### **PreparedStatement** interface

- The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.  
Example of parameterized query:  
String sql="insert into emp values(?,?,?)";
- We are passing parameter (?) for the values. Its value will be set by calling the setter methods(for example setInt(),setString( ) methods) of PreparedStatement.
- **Improves performance:** The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.



***How to get the instance of PreparedStatement?***

- The prepareStatement() method of Connection interface is used to return the object of PreparedStatement. Syntax:

```
public PreparedStatement prepareStatement(String query)throws SQLException
{
}
}
```

**Methods of PreparedStatement interface**

The important methods of PreparedStatement interface are given below:

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

**Callable statement**

- This interface extends PreparedStatement interface and provides support for both input as well as output parameter.
- Callable statement represents the stored procedure.
- Stored procedure: it is a subroutine used by applications to access data from a database.

**➤ JDBC drivers**

- JDBC driver is a software component that enables java application to interact with database.
- It implements the defined interfaces in the JDBC API for interacting with your database server.

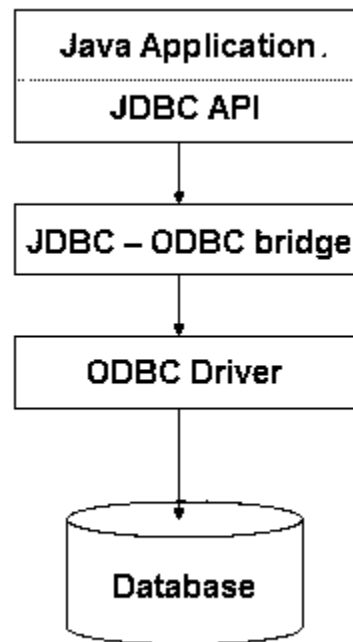
- For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.
- The *Java.sql* package that ships with JDK contains various classes with their behaviours defined and their actual implementations are done in third-party drivers. Third party vendor's implements the *java.sql.Driver* interface in their database driver.

### JDBC Drivers Types:

JDBC driver implementations differ because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which are explained below:

#### Type 1: JDBC-ODBC Bridge Driver:

- In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.
- When Java first came out, this 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.



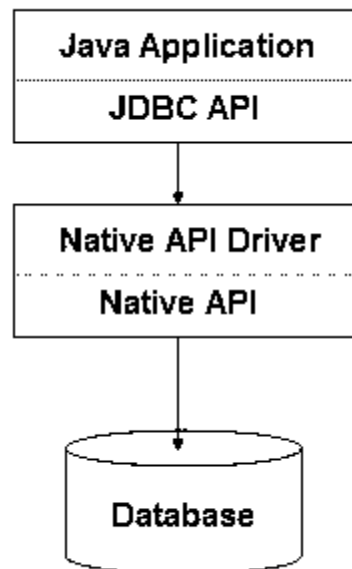
Type 1: JDBC-ODBC Bridge

The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

#### Type 2: JDBC-Native API:

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database.

- These drivers 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.
- If we change the Database we have to change the native API as it is specific to a database and they are mostly obsolete now but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

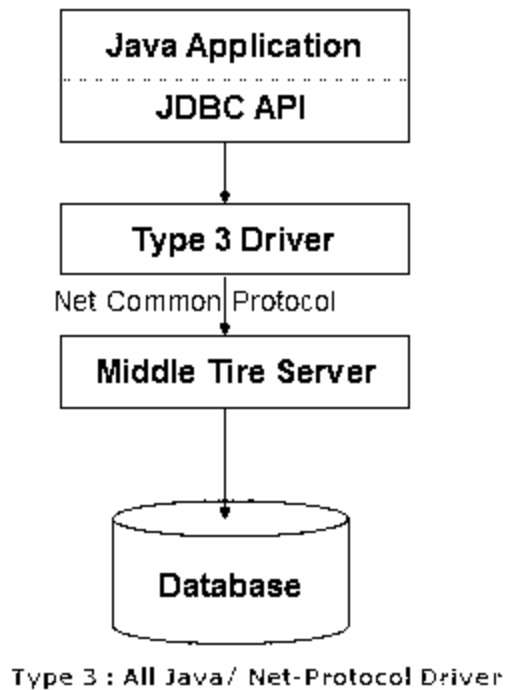


**Type 2: Native API / Partly Java Driver**

The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

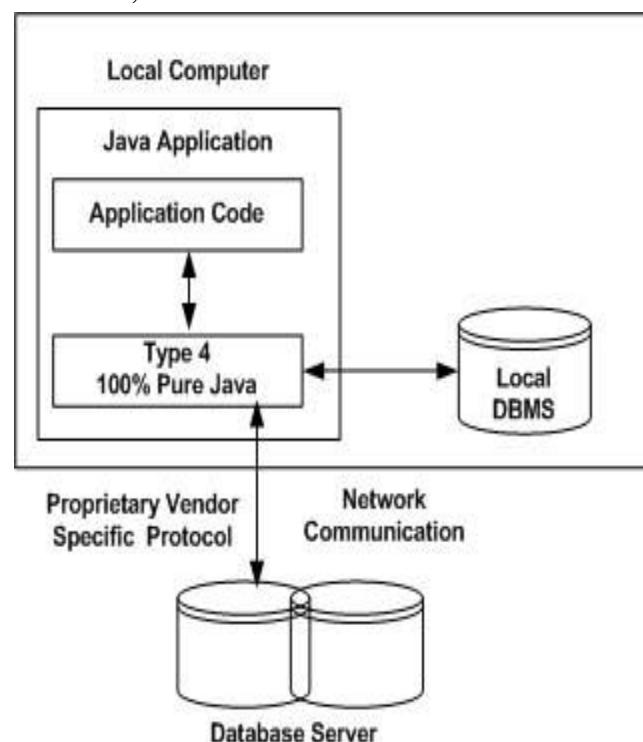
### **Type 3: JDBC-Net pure Java:**

- In a Type 3 driver, a three-tier approach is used to accessing databases. The JDBC clients use standard network sockets to communicate with an middleware application server.
- The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.
- 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.
- We can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, we need some knowledge of the application server's configuration in order to effectively use this driver type.
- Our application server might use a Type 1, 2, or 4 drivers to communicate with the database, understanding the nuances will prove helpful.

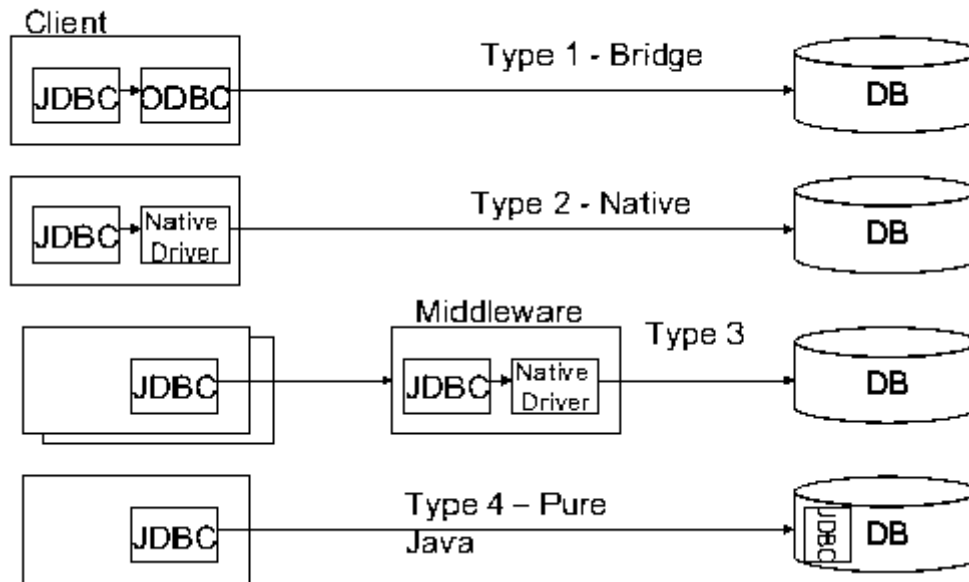


#### Type 4: 100% pure Java:

- In a Type 4 driver, a pure Java-based driver that communicates directly with vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.
- This kind of driver is extremely flexible; you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.



- MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.



### Which Driver should be used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

### Why use JDBC

- Ease of programming
- You can write Java language instead of SQL statements
- Performance improvement
- Able to rollback to the save set of data
- Support batch updates
- Have many optional packages to use

## JDBC : advantage and disadvantage

### Advantage

- **Provide Existing Enterprise Data**

Businesses can continue to use their installed databases and access information even if it is stored on different database management systems

- **Simplified Enterprise Development**

The combination of the Java API and the JDBC API makes application development easy and cost effective

- **Zero Configuration for Network Computers**

No configuration is required on the client side centralizes software maintenance. Driver is written in the Java ,so all the information needed to make a connection is completely defined by the JDBC URL or by a DataSource object. DataSource object is registered with a Java Naming and Directory Interface (JNDI) naming service.

- **Full Access to Metadata**

The underlying facilities and capabilities of a specific database connection need to be understood. The JDBC API provides metadata access that enables the development of sophisticated applications.

## Disadvantage

- The process of creating a connection, always an expensive, time-consuming operation, is multiplied in these environments where a large number of users are accessing the database in short, unconnected operations.
- Creating connections over and over in these environments is simply too expensive.
- When multiple connections are created and closed it affects the performance.

## ❖ Database creation

- **Structured Query Language (SQL)** is a standardized language that allows you to perform operations on a database, such as creating entries, reading content, updating content, and deleting entries.
- SQL is supported by all most any database you will likely use, and it allows you to write database code independently of the underlying database.

### 1. Create Database:

- The CREATE DATABASE statement is used for creating a new database. The syntax is:

**SQL>CREATE DATABASE DATABASE\_NAME;**

Example: The following SQL statement creates a Database named EMP:

**SQL>CREATE DATABASE EMP;**

### 2. Drop Database:

- The DROP DATABASE statement is used for deleting an existing database. The syntax is:

**SQL>DROP DATABASE DATABASE\_NAME;**

**Note:** To create or drop a database you should have administrator privilege on your database server. Be careful, deleting a database would loss all the data stored in database.

### 3. Create Table:

- The CREATE TABLE statement is used for creating a new table. The syntax is:  
**SQL>CREATE TABLE** table\_name  
(  
    column\_name column\_data\_type,  
    column\_name column\_data\_type,  
    column\_name column\_data\_type  
    ...  
);

Example: The following SQL statement creates a table named Employees with four columns:

```
SQL>CREATETABLEEmployees
(
    id INT NOT NULL,
    age INT NOT NULL,
    first VARCHAR(255),
    last VARCHAR(255),
    PRIMARY KEY ( id )
);
```

### 4. Drop Table:

- The DROP TABLE statement is used for deleting an existing table. The syntax is:

```
SQL> DROP TABLE table_name;
```

Example: The following SQL statement deletes a table named employees:

```
SQL> DROP TABLE Employees;
```

### 5. INSERT Data:

- The syntax for INSERT looks similar to the following, where column1, column2, and so on represent the new data to appear in the respective columns:

```
SQL>INSERT INTO table_name VALUES(column1, column2,...);
```

- Example:The following SQL INSERT statement inserts a new row in the Employees database created earlier:

```
SQL>INSERT INTOEmployeesVALUES(100,18,'Zara','Ali');
```

### 6. SELECT Data:

- The SELECT statement is used to retrieve data from a database. The syntax for SELECT is:

```
SQL> SELECT column_name, column_name,...
      FROM table_name
      WHERE conditions;
```



- The WHERE clause can use the comparison operators such as =, !=, <, >, <=, and >=, as well as the BETWEEN and LIKE operators.

**Example:** The following SQL statement selects the age, first and last columns from the Employees table where id column is 100:

```
SQL> SELECT first,last, age
      FROM Employees
      WHERE id =100;
```

## 7. UPDATE Data:

- The UPDATE statement is used to update data. The syntax for UPDATE is:

```
SQL> UPDATE table_name
      SET column_name = value, column_name = value,...
      WHERE conditions;
```

- The WHERE clause can use the comparison operators such as =, !=, <, >, <=, and >=, as well as the BETWEEN and LIKE operators.

Example: The following SQL UPDATE statement changes the age column of the employee whose id is 10

```
SQL> UPDATE Employees SET age=20 WHERE id=100;
```

## 8. DELETE Data:

- The DELETE statement is used to delete data from tables. The syntax for DELETE is:

```
SQL> DELETE FROM table_name WHERE conditions;
```

- The WHERE clause can use the comparison operators such as =, !=, <, >, <=, and >=, as well as the BETWEEN and LIKE operators.

## ❖ Creating JDBC Application:

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

- **Import the packages** . Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.\** will suffice.
- **Register the JDBC driver** . Requires that you initialize a driver so you can open a communications channel with the database.
- **Open a connection** . Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query** . Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set** . Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment** . Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

## Steps to create JDBC application

There are 5 steps to connect any java application with the database in java using JDBC.

They are as follows:

- Register the driver class
- Creating connection
- Creating statement
- Executing queries
- Closing 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.

**Syntax :**

```
public static void forName(String className)throws ClassNotFoundException
```

**Example** to register the OracleDriver class

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

### 2) Create the connection object

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

**Syntax:**

- public static Connection **getConnection**(String url)throws SQLException
- **public static** Connection **getConnection**(String **url**,String **name**,String **password**) **throws** SQLException

**Example** to establish connection with the Oracle database

```
Connection con=DriverManager.getConnection( "jdbc:oracle:thin:@localhost:1521:xe","system","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.

**Syntax**

```
public Statement createStatement()throws SQLException
```

**Example**

```
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

**Syntax**

**public** ResultSet executeQuery(String sql)**throws** SQLException

**Example**

```
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
{
    System.out.println(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.

**Syntax**

**public void close()**throws SQLException

**Example**

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

**Types of driver and its URL to use in program**

RDBMS	JDBC driver name	URL format
<b>MySQL</b>	<b>com.mysql.jdbc.Driver</b>	<b>jdbc:mysql://hostname/ databaseName</b>
<b>Ms access</b>	<b>sun.jdbc.odbc.JdbcOdbcDriver</b>	<b>jdbc:odbc:databasename</b>
ORACLE	oracle.jdbc.driver.OracleDriver	<b>jdbc:oracle:thin:</b> @hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	<b>jdbc:db2:</b> hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	<b>jdbc:sybase:Tds:</b> hostname: port Number/databaseName

## Example to connect to the mysql database

For connecting java application with the mysql database, you need to follow 5 steps to perform database connectivity.

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/mydb**

where **jdbc** is the API, **mysql** is the database, **localhost** is the server name on which mysql is running,

we may also use IP address, **3306** is the port number and **mydb** is the database name. We may use any database, in such case, you need to replace the mydb with your database name.

3. **Username:** The default username for the mysql database is **root**.
4. **Password:** Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

1. create database mydb;
2. use mydb;
3. create table emp(id **int(10)**,name varchar(**40**),age **int(3)**);

## Example to Connect Java Application with mysql database

In this example, mydb is the database name, root is the username and password.

```
import java.sql.*;
class MysqlCon
{
    public static void main(String args[])
    {
        try
        {
            Class.forName("com.mysql.jdbc.Driver");           //step1
```

```
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/mydb","root","root");    //step2

//here mydb is database name, root is username and password

Statement stmt=con.createStatement();                //step3

ResultSet rs=stmt.executeQuery("select * from emp");  //step4

while(rs.next())
System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));

con.close();                                          //step5

}catch(Exception e){ System.out.println(e);}

}

}
```

**To connect java application with the mysql database mysqlconnector.jar file is required to be loaded.**

[download the jar file mysql-connector.jar](#)

Two ways to load the jar file:

1. paste the mysqlconnector.jar file in jre/lib/ext folder
2. set classpath

1) paste the mysqlconnector.jar file in JRE/lib/ext folder:

**Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file**

2) set classpath:

**There are two ways to set the class path:**

a) Temporary :

Open command prompt and write:

C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar,;

b) Permanent path:

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar,; as C:\folder\mysql-connector-java-5.0.8-bin.jar,;.

## JDBC-ODBC bridge

- JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.
- Here we learn how to connect to Microsoft Access database. In order to make the following code workable, we have to set up environment first. Following the steps below:

### Steps to connect using ODBC with MS Access database

1. Click start → go to Control Panel → Administrative Tools → Data Sources → User DSN → add
2. Select Microsoft access Driver → Finish
3. Type Data source Name and press Select. Ok → ok

**The following example shows you how to follow the above steps to load a driver, make a connection, use Statement and PreparedStatement, and insert and query data.**

```
import java.sql.*;
public class TestDBDriver
{
    Connection con;
    Statement stmt;
    ResultSet rs;
    public static void main(String[] args)
    {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con = DriverManager.getConnection(jdbc:odbc:mydb, "", "");
            stmt = con.createStatement();
            //create table
            String sql = "create table book(bid int,bname varchar(20),author varchar(30))";
            stmt.executeUpdate(sql);

            //insert data in table
            String inssql="INSERT INTO book VALUES (1001,'Java', 'Balagurusamy') ";
            stmt.executeUpdate(inssql);

            //display in output
            String selsql="SELECT * from book";
            rs = stmt.executeQuery(selsql);
```

```
while (rs.next())
{
    System.out.print(rs.getInt("bid")+"\t") ;
    System.out.print(rs.getString("bname")+"\t") ;
    System.out.print(rs.getString("author")+"\n") ;
}
stmt.close();
con.close();

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

### ❖ Program using createStatement( )

**Example-1** Write a program to create database in using JDBC.

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

```
Statement stmt=cn.createStatement();
        String sql = "create Database mydb";
        stmt.execute(sql);
        System.out.println("database created") ;

        stmt.close() ;
        cn.close() ;

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



**Output:** database created

### Example-2

**Write a JDBC program to create table having following structure.**

<b>id</b>	<b>name (20)</b>
-----------	------------------

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

```
public class CreateTable
```

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

```
{
    try
    {
        Class.forName("com.mysql.jdbc.Driver");
        Connection
cn=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","");
```

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

```
    stmt.execute("drop table stud"); //if table stud is already exit
```

```
    String sql = "create table stud(id int,name varchar(20))";
```

```
    stmt.execute(sql);
```

```
    System.out.println("Table is created") ;
```

```
    stmt.close() ;
```

```
    cn.close() ;
```

```
}
```

```
catch(Exception e)
```

```
{
```

```
    System.out.println(e);
```

```
}
```

```
}
```

```
}
```

**Output:** Table is created

### Example -3

**Write a JDBC program to insert data into table.**

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

```
public class insert
```

```
{
```

```
    public static void main(String[] args)
```

```
{
```

```
    try
```

```
{
```

```
Class.forName("com.mysql.jdbc.Driver");
Connection
cn=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","");
```

```
Statement stmt=cn.createStatement();
String sql = "INSERT INTO stud VALUES (1, 'Rita')";
stmt.execute(sql);
```

```
String sql = "INSERT INTO stud VALUES (2, 'Ram')";
stmt.execute(sql);
```

```
System.out.println("Row is inserted") ;
```

```
ResultSet rs=stmt.executeQuery("select DISTINCT * from stud ");
while(rs.next())
{
    System.out.print(rs.getInt("id")+"\t");
    System.out.print(rs.getString("name")+"\n");
}
rs.close() ;
stmt.close() ;
cn.close() ;
}
catch(Exception e)
{
    System.out.println(e);
}
}
```

### **Output:**

Row is inserted

```
1 Rita
2 Ram
```

### **Example -4 Write a JDBC program to delete data from table.**

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

```

        Class.forName("com.mysql.jdbc.Driver");
        Connection
cn=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","");

```

**Statement** stmt=cn.createStatement();

```
String sql = "delete from stud " + "where id=2";
```

```
stmt.execute(sql);
```

```
System.out.println("Row is deleted") ;
```

```
ResultSet rs=stmt.executeQuery("select DISTINCT * from stud ");
```

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

```
{
```

```
    System.out.print(rs.getInt("id")+"\t"); // you can use 1 instead of id,where
1 is column index
```

```
    System.out.print(rs.getString("name")+"\n");
```

```
}
```

```
rs.close() ;
```

```
stmt.close() ;
```

```
cn.close() ;
```

```
}
```

```
catch(Exception e)
```

```
{
```

```
    System.out.println(e);
```

```
}
```

```
}
```

```
}
```

**Output :**

```
Row is deleted
```

```
1      Rita
```

**Example -5** Write a JDBC program to update data from table.

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

```
public class update
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        try
```

```
        {
```

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

```
            Connection
```

```
cn=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","");
```

**Statement** stmt=cn.createStatement();

```

        int up=stmt.executeUpdate("update stud set name='Sita' where id=1");
        if(up>0)
        {
            System.out.println("Record Sucessfully Updated...!!!");
        }
        ResultSet rs=stmt.executeQuery("select DISTINCT * from stud ");
        while(rs.next())
        {
            System.out.print(rs.getInt("id")+"\t");
            System.out.print(rs.getString("name")+"\n");
        }
        rs.close() ;
        stmt.close() ;
        cn.close() ;
    }
    catch(Exception e)
    {
        System.out.println(e);
    }
}
}

```

**Output :**

Record Sucessfully Updated...!!!  
 1   Sita

**Example -6          Write a JDBC program to display record from table.**

```

import java.sql.*;
public class selectdata
{
    public static void main(String args[])
    {
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            Connection
cn=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","");
Statement stmt=cn.createStatement();
            System.out.println(" Table data :");
            ResultSet rs=stmt.executeQuery("select distinct * from stud");
            while(rs.next())
            {
                System.out.print(rs.getInt("id")+ "\t");
                System.out.print(rs.getString("name")+"\n");
            }
        }
    }
}

```

```

    }
}
catch(Exception e)
{
    System.out.println(e);
}
}
}

```

### Output ;

Table data :

1      Sita

### ❖ Program using `prepareStatement()`

#### Example -7

**Write JDBC program to insert data using `PreparedStatement` interface.**

```

import java.sql.*;
public class insert
{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            Connection
cn=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","");
String sql = "insert into stud(id, name) values(?, ?)";
            PreparedStatement pstmt=cn.prepareStatement(sql);
            pstmt.setInt(1, 10); // set input parameter 1 for id,value 100
            pstmt.setString(2, "dhara"); // set input parameter 2 for name ,value dhara
            pstmt.executeUpdate();
            ResultSet rs=pstmt.executeQuery("select DISTINCT * from stud ");
            while(rs.next())
            {
                System.out.print(rs.getInt("id")+"\t");
                System.out.println(rs.getString("name")+"\t");
            }
            pstmt.close() ;
            cn.close() ;
        }
        catch(Exception e)
        {
            System.out.println(e);

```

```

    }
}
}
Output :
    10    dhara

```

### Example -8

**Write JDBC program to delete data from table using PreparedStatement interface.**

```

import java.sql.*;
public class delete

{
    public static void main(String[] args)
    {

        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            Connection
cn=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","");
            String sql="delete from stud where id=?";
PreparedStatement pstmt=cn.prepareStatement(sql);
            pstmt.setInt(1,10);        //10 is id you want to delete
            pstmt.executeUpdate();
            ResultSet rs=pstmt.executeQuery("select DISTINCT * from stud ");
            while(rs.next())
            {
                System.out.print(rs.getInt("id")+"\t");
                System.out.println(rs.getString("name")+"\t");
            }
            rs.close() ;
            pstmt.close() ;
            cn.close() ;

        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}

```

**Example -9**

**Write JDBC program to update data from table using PreparedStatement interface.**

```
import java.sql.*;
public class updatepre
{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            Connection
cn=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb1","root","");
            String sql= "UPDATE stud SET name = ? WHERE id = ?";
PreparedStatement pstmt = cn.prepareStatement(sql);
            pstmt.setString(1, "abc");    // name =abc
            pstmt.setInt(2, 10);          //id=10
            // execute insert SQL statement
            int up=pstmt.executeUpdate();
            if(up>0)
            {
                System.out.println("Record Sucessfully Updated...!!!");
            }
            ResultSet rs=pstmt.executeQuery("select DISTINCT * from stud ");
            while(rs.next())
            {
                System.out.print(rs.getInt("id")+"\t");
                System.out.print(rs.getString("name")+"\n");
            }
            rs.close() ;
            pstmt.close() ;
            cn.close() ;

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

**Output:**

Record Sucessfully Updated...!!!  
10     abc



Example 10 :

Write a program which creates GUI and performs following operations:

1) Insert 2) delete 3) Update

```
import java.awt.*;
import javax.swing.*;
import java.sql.*;
import java.awt.event.*;

public class textdata extends JApplet implements ActionListener
{
    JTextField jtf1=new JTextField(10);
    JTextField jtf2=new JTextField(10);
    JLabel jlb1=new JLabel("id :",JLabel.CENTER);
    JLabel jlb2=new JLabel("name:",JLabel.CENTER);
    JLabel jlb3=new JLabel("      ");
    JLabel jlb4=new JLabel("      ");
    JButton jbinst=new JButton("Insert");
    JButton jbdlt=new JButton("Delete");
    JButton jbrst=new JButton("Reset");
    JButton jbupdt=new JButton("Update");
    Statement stmt;
    ResultSet rs;
    public void init()
    {
        jbinst.addActionListener(this);
        jbdlt.addActionListener(this);
        jbrst.addActionListener(this);
        jbupdt.addActionListener(this);

        JPanel jp=new JPanel();
        jp.setLayout(new GridLayout(0,2));
        jp.add(jlb1);
        jp.add(jtf1);
        jp.add(jlb2);
        jp.add(jtf2);
        jp.add(jbinst);
        jp.add(jbdlt);
        jp.add(jbrst);
        jp.add(jbupdt);
        jp.add(jlb3);
        jp.add(jlb4);

        add(jp);
        initializeDB();
    }
}
```

```

}
private void initializeDB()
{
try
{
    Class.forName("com.mysql.jdbc.Driver");
    Connection
    cn=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root"
    , "");
    System.out.println("database connection");
    stmt=cn.createStatement();
}
catch(Exception e)
{
    System.out.println(e);
}
}

public void actionPerformed(ActionEvent e)
{
String s1,s2;

if(e.getActionCommand().equals("Insert"))
{

    s1= jtf1.getText();
    s2= jtf2.getText();
    try
    {
        int      in=stmt.executeUpdate("INSERT      INTO      stud      VALUES
        (""+s1+"",""+s2+"")");
        if(in>0)
            System.out.println("inserted") ;
        String qs="select distinct* from stud";
        rs=stmt.executeQuery(qs);
        if(rs.next())
        {
            String st1=s1;//rs.getString(1);
            String st2=s2;//rs.getString(2);
            jlb3.setText("id is: "+st1);
            jlb4.setText("name is: "+st2);
        }
    }
    catch(SQLException ex)
    {

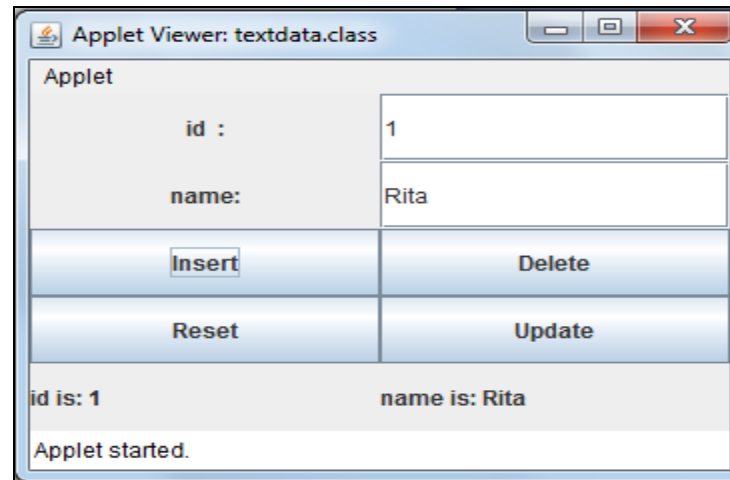
```

```
        ex.printStackTrace();
    }
}
if(e.getActionCommand().equals("Delete"))
{
    try
    {
        String sql = "delete from stud " + "where id="+jtf1.getText();
        stmt.execute(sql);
        JOptionPane.showMessageDialog(null, "Record is deleted!!!");
    }
    catch(SQLException ex)
    {
        ex.printStackTrace();
    }
}

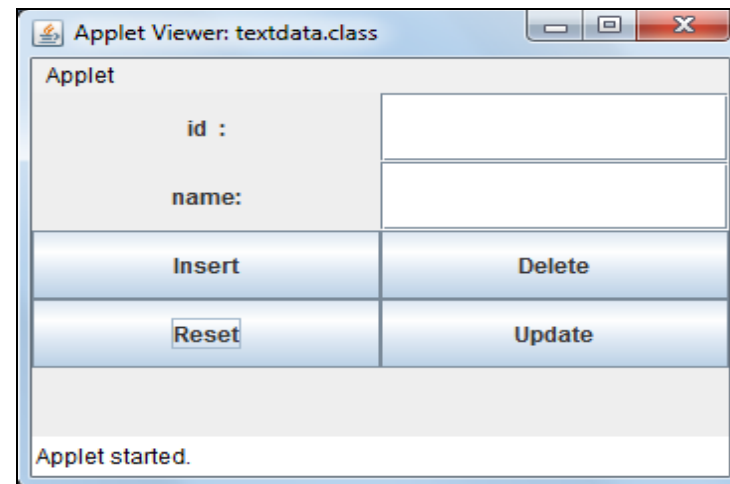
if(e.getActionCommand().equals("Update"))
{
    try
    {
        String sql="update stud set name='"+jtf2.getText()+"' where id =
        "+jtf1.getText()+"";
        stmt.execute(sql);
        JOptionPane.showMessageDialog(null, "Record is Updated!!!");
    }
    catch(SQLException ex)
    {
        ex.printStackTrace();
    }
}
if(e.getActionCommand().equals("Reset"))
{
    jtf1.setText("");
    jtf2.setText("");
    jlb3.setText("");
    jlb4.setText("");
}

}

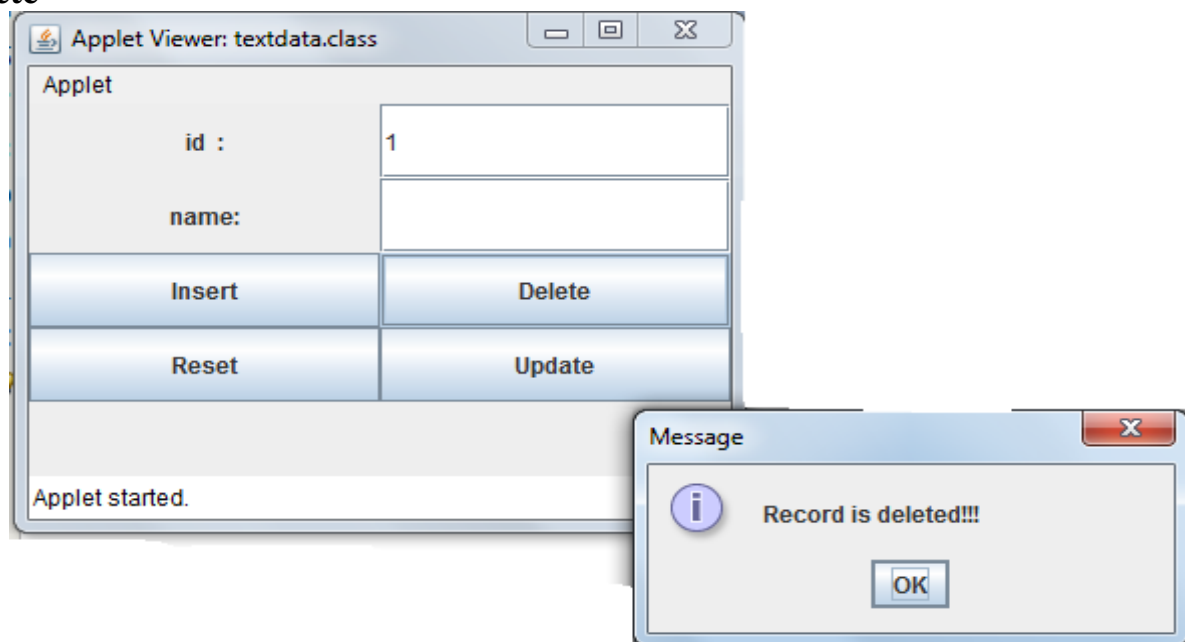
}
```

**Output:  
Insert**

The screenshot shows the 'Applet Viewer: textdata.class' window. It contains a form with two input fields: 'id :' with the value '1' and 'name:' with the value 'Rita'. Below the fields are four buttons: 'Insert', 'Delete', 'Reset', and 'Update'. The 'Insert' button is highlighted. At the bottom, a status bar displays 'id is: 1' and 'name is: Rita', and a message box says 'Applet started.'

**Reset**

The screenshot shows the 'Applet Viewer: textdata.class' window. The 'id :' and 'name:' fields are now empty. The 'Reset' button is highlighted. The status bar at the bottom still displays 'id is: 1' and 'name is: Rita', and the message box says 'Applet started.'

**Delete**

The screenshot shows the 'Applet Viewer: textdata.class' window. The 'id :' field contains '1' and the 'name:' field is empty. The 'Delete' button is highlighted. A 'Message' dialog box is overlaid on the applet, displaying an information icon, the text 'Record is deleted!!!', and an 'OK' button. The status bar at the bottom of the applet window still displays 'id is: 1' and 'name is: Rita', and the message box says 'Applet started.'

**Example -11**

**Write a program which display product detail when user selects any product.**

```
import java.sql.*;
import java.util.*;
import javax.swing.*;
import java.awt.event.*;
public class product extends JApplet implements ActionListener
{
    /**
     create table product(pid int, productname varchar(25), price int);
     insert into product values(1, 'Laptop', 35000);
     insert into product values(2, 'SmartPhone', 15000);
     insert into product values(3, 'LED TV 48 Inch', 50000);
     insert into product values(4, 'Iphone', 60000);
     insert into product values(5, 'Tablet', 10000);
    */

    ArrayList<String> list = new ArrayList<>();
    JComboBox jcb;
    Connection cn;
    Statement stmt;
    ResultSet rs;
    JPanel pc = new JPanel();
    JLabel jlb = new JLabel("Select Product");
    JLabel jlb1 = new JLabel("Product = ");
    JLabel jlb2 = new JLabel("Price = ");
    JLabel jlproduct = new JLabel();
    JLabel jlprice = new JLabel();
    public void init()
    {
        initializeDB();
        try
        {
            list.add("Select Product");
            ResultSet rs = stmt.executeQuery("select * from product");
            while(rs.next())
            {
                list.add(rs.getString("productname"));
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

```

    }
    pc.setLayout(null);
    jlb.setBounds(15, 25, 100, 25);
    jcb = new JComboBox(list.toArray());
    jcb.setBounds(110, 25, 150, 25);
    jcb.addActionListener(this);
    jlb1.setBounds(15,60,70,25);
    jlproduct.setBounds(80,60,160,25);
    jlb2.setBounds(180,60,50,25);
    jlprice.setBounds(230,60,70,25);
    pc.add(jlb);
    pc.add(jcb);
    pc.add(jlb1);
    pc.add(jlb2);
    pc.add(jlproduct);
    pc.add(jlprice);
    add(pc);
    setSize(325, 245);
}

private void initializeDB()
{
    try {
        Class.forName("com.mysql.jdbc.Driver");
        Connection
cn=DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","root","");
        System.out.println("database connection");
        stmt=cn.createStatement();
    } catch(Exception e){
        System.out.println(e);
    }
}

public void actionPerformed(ActionEvent e)
{
    JComboBox cb = (JComboBox)e.getSource();
    String proName = (String)cb.getSelectedItem();
updateLabel(proName);
}

public void updateLabel(String proName)
{
    try
    {
        ResultSet rs = stmt.executeQuery("select * from product where
productname='"+proName+"'");

```

```
        while(rs.next())
        {
            jlproduct.setText(proName);
            jlprice.setText(rs.getString("price"));
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
```

**Output:**