

---

*This will cover java Database programming + chapters of “Advanced Java Programming” –Unit: 4 (BSc. CSIT, TU)*

---

## **Database Connectivity**

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. Database management systems allow for the access and storage of data without concern for the internal representation of data.

Today’s most popular database systems are relational databases. A language called SQL—pronounced “sequel,” or as its individual letters—is the international standard language used almost universally with relational databases to perform queries (i.e., to request information that satisfies given criteria) and to manipulate data.

Some popular relational database management systems (RDBMSs) are Microsoft SQL Server, Oracle, Sybase, IBM DB2, Informix, PostgreSQL and MySQL. The JDK now comes with a pure-Java RDBMS called Java DB—Oracles’s version of Apache Derby.

Java programs communicate with databases and manipulate their data using the Java Database Connectivity (JDBC™) API. A JDBC driver enables Java applications to connect to a database in a particular DBMS and allows you to manipulate that database using the JDBC API.

### **What is JDBC?**

JDBC stands for Java Database Connectivity which is a technology that allows Java applications to interact with relational databases and execute SQL queries against the databases

JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

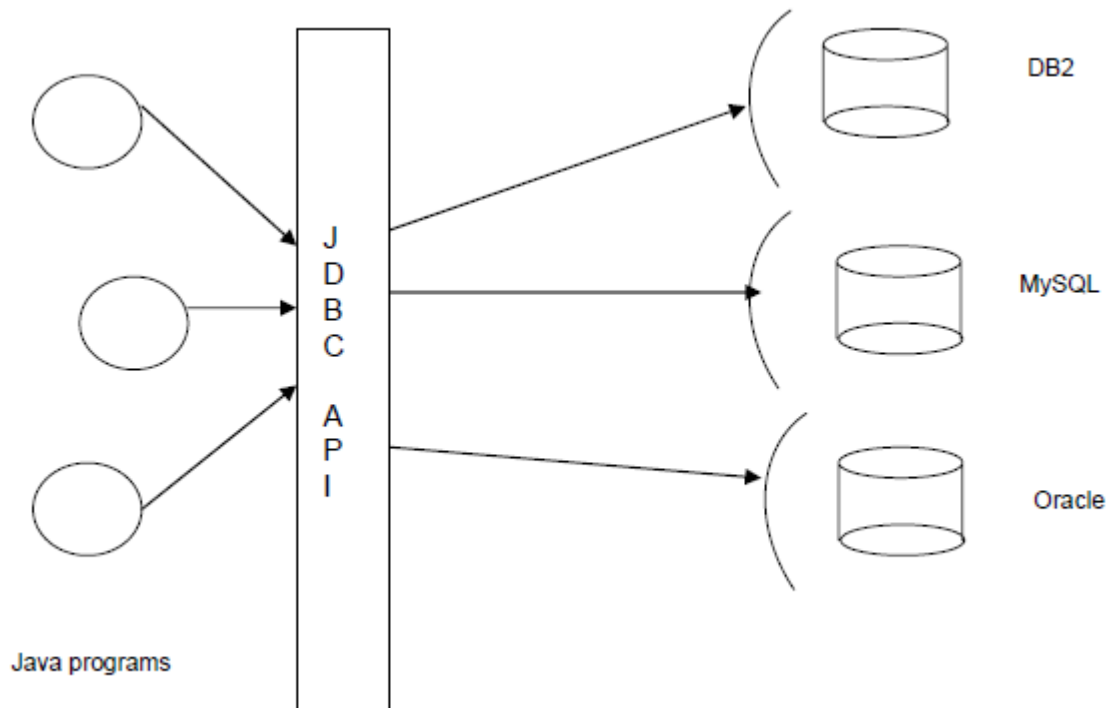
JDBC helps to write Java applications that manage these three programming activities:

- ✓ 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 your query

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executable, such as –

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executable are able to use a JDBC driver to access a database, and take advantage of the stored data.



*Figure 1 JDBC and Databases*

The above figure should tell you the complete story. Java applications simply use JDBC API to interact with any database.

## Typical Uses of JDBC

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

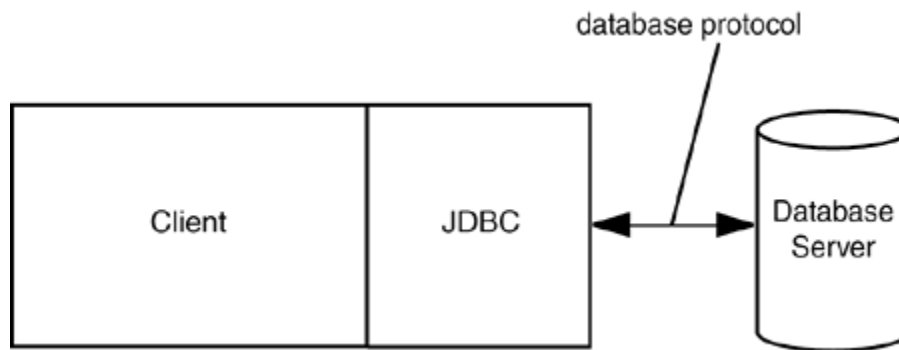


Figure 2: Traditional Client/Server application

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.

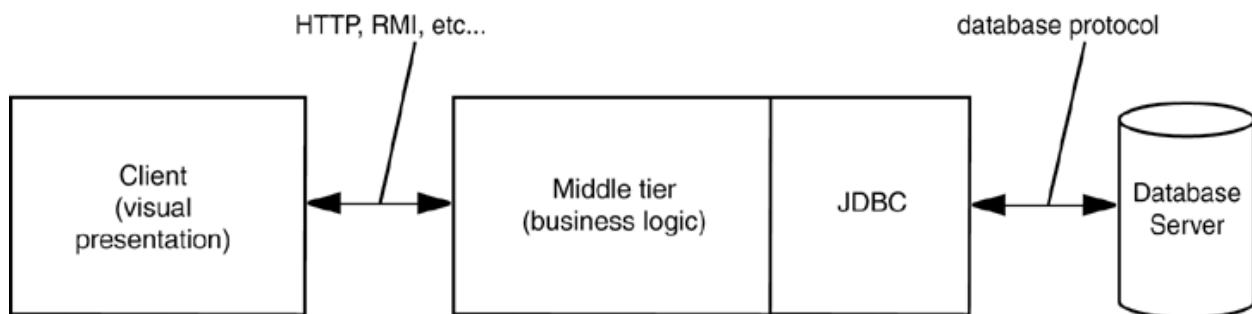


Figure 3: A three-tier application

## How JDBC talks with Databases?

- Databases are proprietary applications that are created by noted companies like Oracle, Microsoft and many more. These could be developed in any language like C, C++ etc by these vendors.
- Every database will expose something called Driver through which one can interact with database for SQL operations.
- A driver is like a gateway to the database.
- Therefore, **for any Java application to work with databases, it must use the driver of that database.**

## Database Drivers

- ✗ A driver is not a hardware device. It's a software program that could be written in any language.
- ✗ Every database will have its own driver program.
- ✗ Given a driver program of a particular database, the challenge is how to use it to talk with database. To understand this, we need to know the different types of drivers.

***ODBC** (Open Database Connectivity) is a standard database access method developed by Microsoft Corporation. ODBC makes it possible to access data from any application, regardless of which database management system (DBMS) is handling the data*

*Before JDBC, ODBC API was the database API to connect and execute 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).*

- *ODBC is hard to learn.*
- *ODBC has a few commands with lots of complex options. The preferred style in the Java programming language is to have simple and intuitive methods, but to have lots of them.*
- *ODBC relies on the use of void\* pointers and other C features that are not natural in the Java programming language.*
- *An ODBC-based solution is inherently less safe*

There are basically 4 different types of database drivers as described below:

### 1. JDBC-ODBC Bridge Driver (Type -1 driver)

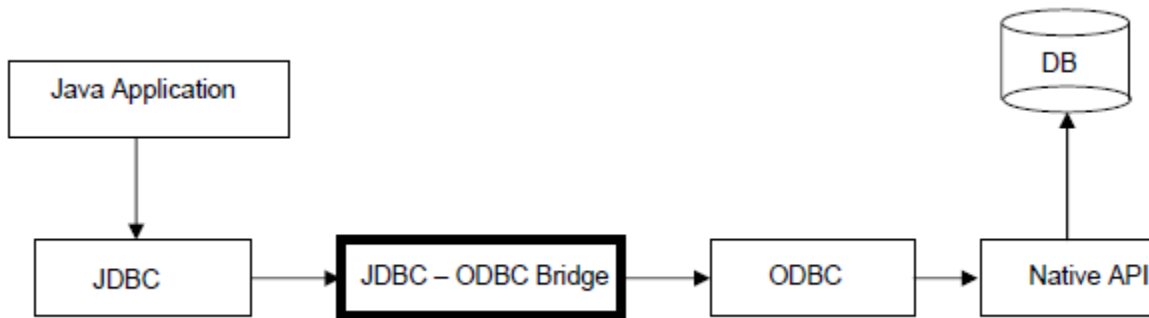


Figure 4: Type -1 Driver

- ✎ This is also called as Type-1 driver
- ✎ As you can notice from the above figure, this driver translates the JDBC calls to ODBC calls in the ODBC layer. The ODBC calls are then translated to the native database API calls. Because of the multiple layers of indirection, the performance of the application using this driver suffers seriously. This type of driver is normally used for training purposes and never used in commercial applications. The only good thing with this driver is that you can access any database with it.
- ✎ In simple words, with this driver for JDBC to talk with vendor specific native API, following translations will be made
 

**JDBC - > ODBC -> Native**

  - ◆ It's a 2 step process.
- ✎ The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

## 2. Partly Java and Partly Native Driver (Type -2 driver)

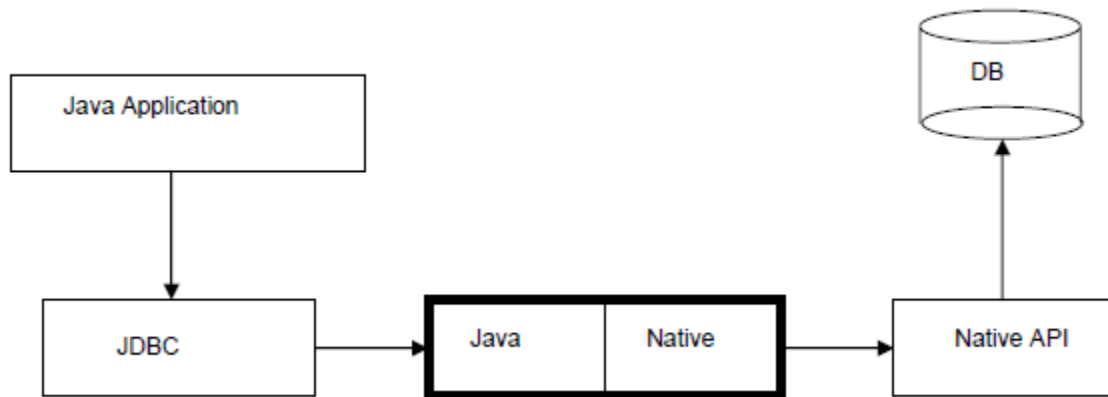


Figure 5: Type -2 Driver

- ✎ This is also called as Type-2 driver
- ✎ As the name suggests, this driver is partly built using Java and partly with vendor specific API
- ✎ With this driver, the JDBC calls are translated to vendor specific native calls in just one step. It completely eliminates the ODBC layer. Because of this ODBC layer elimination, the applications using this driver perform better.
- ✎ The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

## 3. Intermediate Database access Driver Server (Type -3 Driver)

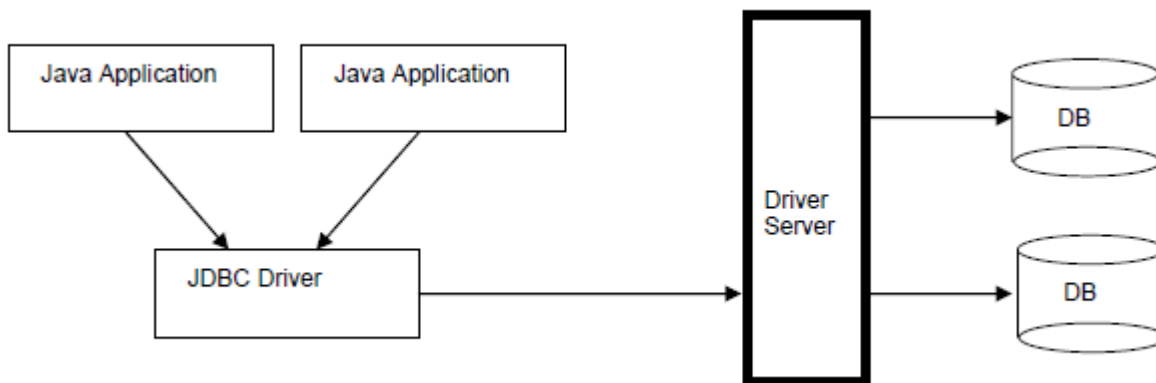


Figure 6: Type- 3 Driver

- ✎ This is also called as Type-3 driver
- ✎ If you look at the Type-2 driver configuration, we only access one database at a time. However there will be situations where multiple Java programs need to access to multiple databases. This is where Type-3 driver is used.
- ✎ It acts as a middleware for the applications to access several databases. The Type-3 configuration internally may use Type-2 configuration to connect to database

✎ IDS JDBC Driver is an example

#### 4. Pure Java Drivers (Type -4 Drivers)

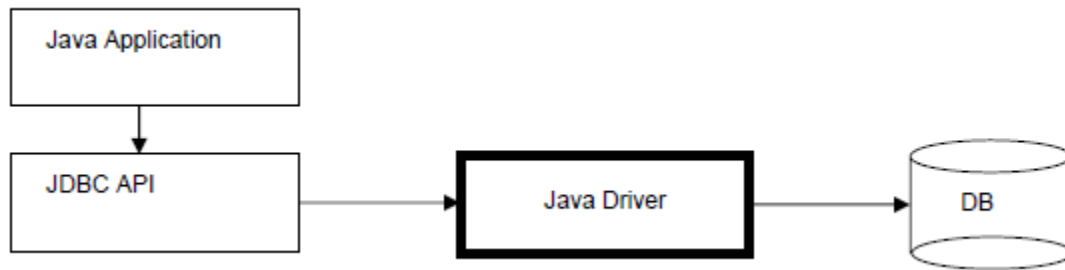


Figure 7: Type - 4 Driver

- ✎ These are called as Type-4 drivers.
- ✎ Because of the widespread usage of Java, to facilitate easy and faster access to databases from Java applications, database vendors built the driver itself using Java like good friends helping each other. This is really cool as it completely eliminates the translation process.
- ✎ JDBC is Java based technology, and with the driver also written in Java, it can directly invoke the driver program as if it were any other Java program.
- ✎ This is the driver that all the J2EE applications use to interact with databases.
- ✎ Because this is a Java to Java interaction, this driver offers the best performance.
- ✎ MySQL's Connector/J driver is a Type 4 driver.

#### 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.

#### Driver JAR Files

- ✓ You need to obtain the JAR file in which the driver for your database is located (e.g. for Derby database, you need the file derbyclient.jar)
- ✓ Include the driver JAR file on the class path
- ✓ Launch programs from the command line with



```
java -classpath .;driverJar ProgramName
```

## JDBC API

JDBC technology is nothing but an API. It is a set of classes and interfaces that our Java programs use to execute the SQL queries against the database. The fundamental idea behind JDBC is that, Java programs use JDBC API, and JDBC API will in turn use the driver to work with the database.

Therefore, to work with MySQL database, we need to configure the JDBC with all the MySQL information. This information is nothing but:

- ◆ Name of the MySQL driver class
- ◆ URL of the database schema.

In simple words, for a Java program to work with the database, we first need to configure JDBC with the above database info and then make the Java program use the JDBC.

Few class and interfaces of JDBC API are as follows:

Class/Interface	Description
DriverManager	This classes is used for managing the Drivers
Connection	It's an interface that represents the connection to the database
Statement	Used to execute static SQL statements
PreparedStatement	Used to execute dynamic SQL statements
CallableStatement	Used to execute the database stored procedures
ResultSet	Interface that represents the database results
ResultSetMetaData	Used to know the information about a table
DatabaseMetadata	Used to know the information about the database
SQLException	The checked exception that all the database classes will throw.

*The JDBC™ 4.3 ( Released in September 21, 2017) API includes both the java.sql package, referred to as the JDBC core API, and the javax.sql package, referred to as the JDBC Optional Package API. This complete JDBC API is included in the Java™ Standard Edition (Java SE™), version 7. The javax.sql package extends the functionality of the JDBC API from a client-side API*

to a server-side API, and it is an essential part of the Java™ Enterprise Edition (Java EE™) technology.

JDBC programming is the simplest of all. There is a standard process we follow to execute the SQL queries. Following lists the basic steps involved in any JDBC program.

- Import necessary packages
- Load and Register the Driver
- Establish the connection to the database
- Create the statements( using Statement/ PreparedStatement/ CallableStatement)
- Execute the statements
- Process the results
- Close the statements
- Close the connection.

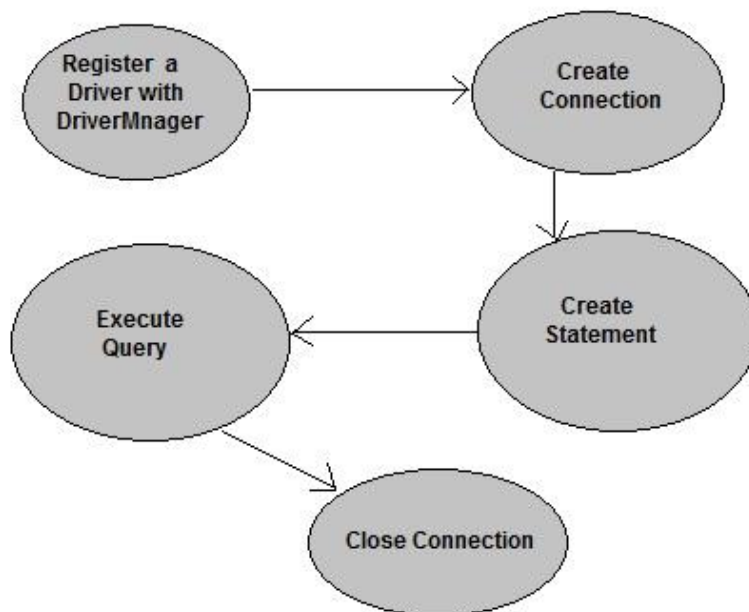


Figure 8: Steps to connect database in java

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

#### Syntax of forName() method

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

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

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

#### Syntax of getConnection() method

1) **public static** Connection getConnection(String url)**throws** SQLException

2) **public static** Connection getConnection(String url,String name,String passwd)  
**throws** SQLException

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

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

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

#### Syntax of createStatement() method

**public** Statement createStatement()**throws** SQLException

#### Example to create the statement object

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

- 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 of executeQuery() method

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

Example to execute query

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

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

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

**Syntax of close() method**

**public void** close()**throws** *SQLException*

Example to close connection

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

## Database URLs

### Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods –

*getConnection(String url)*

*getConnection(String url, Properties prop)*

*getConnection(String url, String user, String password)*

Here each form requires a database URL. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: port Number/databaseName

All the highlighted part in URL format is static and you need to change only the remaining part as per your database setup.

When connecting to a database, you must use parameters like

- host names,
- port numbers, and
- database names.

**Syntax:** *jdbc:subprotocol:other\_parameters*

Subprotocol selects the specific driver for connecting to the database.

The format for the other parameters depends on the subprotocol used.

Examples (database name is Demo)

`jdbc:derby://localhost:1527/Demo;create=true`

`jdbc:mariadb://localhost:3306/Demo`

### Connecting to MySQL Database via JDBC

- At first, Configure MySQL Database server in your PC.
- Configure MySQL Server properties
  - NetBeans IDE comes bundled with support for the MySQL RDBMS. Before you can access the MySQL Database Server in NetBeans IDE, you must configure the MySQL Server properties.
  - Right-click the Databases node in the Services window and choose Register MySQL Server to open the MySQL Server Properties dialog box.

**(Window → Services → Database)**

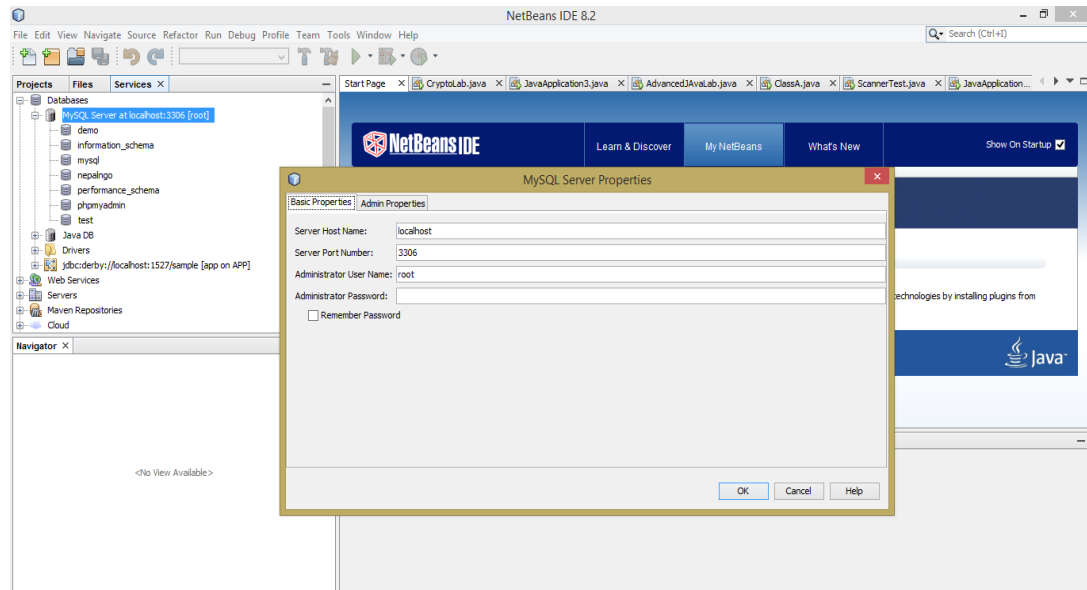


Figure 7: MySQL Database Server Properties

- Confirm that the server host name and port are correct.

Notice that the IDE enters localhost as the default server host name and 3306 as the default server port number.

- Click the “Run Administration Tool” tab at the top of the dialog box.

The Admin Properties tab is then displayed, allowing you to enter information for controlling the MySQL Server.

<http://localhost/phpmyadmin/>

- Create database and tables

### ➤ Create a Java Project

- Add the MySQL JDBC Driver

Because it is a JDBC project and we opt to connect to a MySQL database, we need MySQL JDBC Driver. It is basically a vendor-specific Type 4 driver that bridges the communication gap between our application and the database. If you want to connect to any other database, look for that database-specific JDBC driver. A JDBC

Driver for almost all major databases (Oracle, PostgreSQL, and so on) is available in the relevant vendor sites.

- ✓ Right-click the project name in the Projects tree view → Properties.
- ✓ Select Libraries from the Categories tree view.
- ✓ Click the Add Library button. An Add Library window will appear.

If the MySQL JDBC Driver is not present in the available libraries list, click the Import... button. Then, click the MySQL JDBC Driver from the available libraries list.

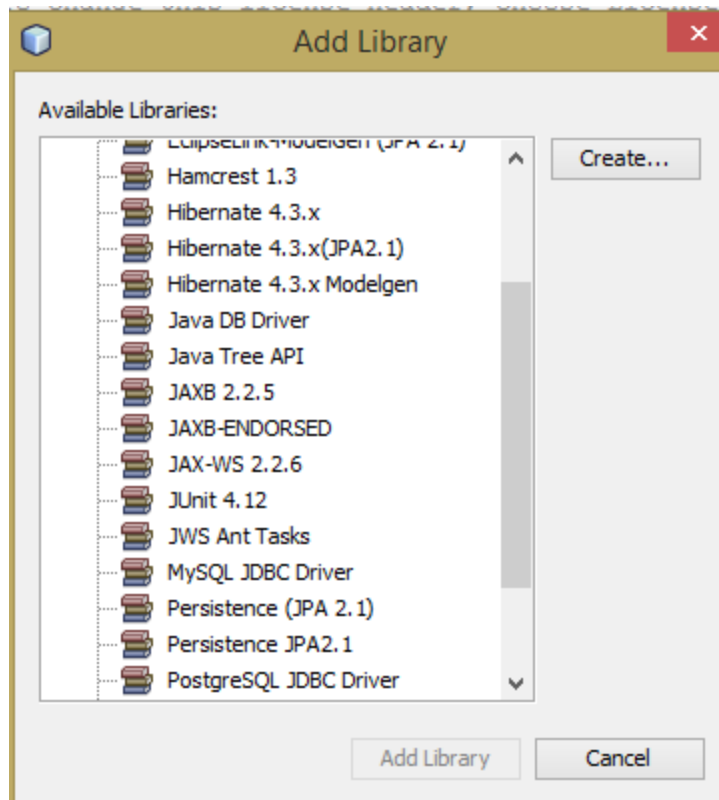


Figure 8: Adding MySQL JDBC Driver in NetBeans

- ✓ *Note: The MySQL JDBC Driver is already shipped with NetBeans. There is no need to install it separately. Other drivers, which may not be shipped with NetBeans, may need to be imported as a separate jar file. In such a case, use the Add JAR/Folder button to import it in the current project.*



- ✓ If the import is successful, the imported jar file will be shown in the projects list (inside Libraries of that project)

For MySQL Connection:

- **Driver class:** The driver class for the mysql database is `com.mysql.jdbc.Driver`.
- **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..
- **Username:** The default username for the mysql database is `root`.
- **Password:** Password is given by the user at the time of installing the mysql database.

### Creating the Application

There are a few things common to Java code in implementing a JDBC application. The package that contains all the required classes and interfaces for JDBC programming is **java.sql**. The exceptions that may occur are handled by the object of the **SQLException** reference defined in the same package.

- Connecting to the database

Database connection is established by creating a Connection object. The Connection is an interface and the object that implements the Connection interface manages the connection between Java and the database. A reference to the Connection object is created with the help of the overloaded static method **getConnection()**, defined in the **DriverManager** class. One of the overloaded **getConnection()** methods takes three parameters.

- ✓ Database URL of the format `jdbc:mysql://hostname:portNumber/databaseName` for MySQL. This format may change when connecting to other databases.
- ✓ Database user name
- ✓ Database password
- Executing the SQL query

Next, we need an object reference to the Statement interface to submit SQL statements to the database.

- Processing the query result

The result of the query fired through the Statement object is contained by an object of the **ResultSet** interface.

### Executing SQL statements

- For SELECT queries, use

```
ResultSet rs = statement.executeQuery("SELECT
* FROM Books")
```

- Process **ResultSet** object using

```
while (rs.next())
{
// look at a row of the result set
}
```

- Inside each row, get field value using

```
String isbn = rs.getString(1); // using column
index
double price = rs.getDouble("Price"); // using
column name
```

Note: You can use execute method to execute arbitrary SQL statements Database column indices start from 1

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:

<b>public ResultSet executeQuery(String sql):</b> is used to execute SELECT query. It returns the object of ResultSet.
--

<b>public int executeUpdate(String sql):</b> is used to execute specified query, it may be create, drop, insert, update, delete etc.
--

<b>public boolean execute(String sql):</b> is used to execute queries that may return multiple results.
---

<b>public int[] executeBatch():</b> is used to execute batch of commands.
---

### Managing Connections, Statements, Result Set

Every Connection object can create one or more Statement objects. You can use the same Statement object for multiple, unrelated commands and queries. A statement has at most one open result set. When you are done using a **ResultSet**, **Statement**, or **Connection**, call the close method immediately (in finally block). The close method of a Statement object automatically closes the associated result set. The close method of the Connection class closes all statements of the connection.

**Example: (Using Statement)**

```

package jdbc_practice;

//import required package

import java.sql.*;

class Example{

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

String className = "com.mysql.jdbc.Driver"; //mysql driver (mysql-connector)

String db_url ="jdbc:mysql://localhost:3306/java2lab";

String db_username="root";

String db_password="";

//registering the driver

Class.forName(className);

//establishing the connection

//getConnection() is the method of DriverManager which will give object of Connection

//remember Connection is an interface we cannot create object directly

Connection con =DriverManager.getConnection(db_url,db_username,db_password);

//Create statements

//remember Statement is also an interface

Statement stmt=con.createStatement();

//Execute query

String my_query= "insert into students values(5,'Ramesh','Thapa',9876007865,'Jhapa)";

//this will insert a row in table "students"

//and affected number of rows will be returned

```

```

int res = stmt.executeUpdate(my_query);

System.out.println(res + "row(s) inserted");

//closing statement

stmt.close();

//closing connection

con.close();

}

}

```

## Reading the Data

This is the most important and widely performed operation. In order to retrieve the data, we use the following method on the statement object:

```
public ResultSet executeQuery (String sql )
```

The above method takes the query and returns a **ResultSet** object that contains all the records returned by the database. We should then iterate over it and display the data. To retrieve the data from the **ResultSet**, there are several get methods available based on the type of the data.

- We use the **executeQuery()** method to execute the SELECT query. This method will return a **ResultSet** object as shown below:

```
ResultSet rs = stmt.executeQuery(myquery);
```

- A resultset object can be imagined like a table of records with a pointer at the beginning of the table as shown below:
- To read the records, we need to move the pointer to the record using the **next()** method until the pointer reaches the end of the records. This is what the while loop does. Once we are in the loop, we need to read the record that the pointer points using the **get** methods.
- The get methods can either take the column number starting from 1 or the column name itself.
- In our example, the column numbers have been used.
- Following are the most commonly used get methods:

```
public String getString() →Used for VARCHAR columns
```

**public int getInt()** →Used for NUMERIC columns

**public Date getDate()** →Used for DATE columns.

- Instead of specifying the column numbers, we can also specify the column names as shown below:

```
String firstName = rs.getString("fname");
```

```
package jdbc_practice;

import java.sql.*;

public class ReadFromTable {

    public static void main(String[] args) {

        String className = "com.mysql.jdbc.Driver";

        String db_url = "jdbc:mysql://localhost:3306/java2lab";

        String db_username = "root";

        String db_password = "";

        try{

            Class.forName(className);

            Connection con;

            Con=
DriverManager.getConnection(db_url,db_username,db_password);

            Statement stmt;

            stmt = con.createStatement();

            String myquery = "SELECT * FROM students";

            ResultSet rs;

            rs = stmt.executeQuery(myquery);

            int rn;

            long phn;

            String fn,ln,adr;
```

```
while(rs.next())
{
    rn =rs.getInt(1);
    fn =rs.getString(2);
    ln = rs.getString(3);
    phn = rs.getLong(4);
    adr = rs.getString(5);

    System.out.println("Roll Number:"+rn+"\t"
        +"First Name:"+fn+"\t"
        +"Last Name:"+ln+"\t"
        +"Phone Number:"+phn+"\t"
        +"Address:"+adr+"\t"
    );

}

stmt.close();

con.close();

} catch(Exception excp){

    System.out.println(excp);

}

}

}
```



**Trick:** If the query is SELECT, use the **executeQuery()** method. For all other queries (CREATE, INSERT, DELETE, UPDATE etc) use the **executeUpdate()** method.

**Commonly used methods of ResultSet interface**

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

<code>public String getString(int columnIndex)</code>	is used to return the data of specified column index of the current row as String.
---	--

<code>public String getString(String columnName)</code>	is used to return the data of specified column name of the current row as String.
---	---

**Some programming tasks:****Task1:**

**Write a java program that allows a user to insert values to a table of particular database (Suppose database is in MySql server ). The program should take the values to insert from console.**

**Task2:**

**WAP using JDBC to display the records from a table of given database (Suppose database is in MySql server ). Assume the following table :**

```
result(roll_no , course_name ,marks_obtained)
```

**The program should read the roll number value from console and display the**

```
//Task1 Code

package jdbc_practice;

import java.sql.*;
import java.util.Scanner;

public class Task1 {

    private final String DRIVER_NAME = "com.mysql.jdbc.Driver";
```

```

private          final          String          DB_URL
="jdbc:mysql://localhost:3306/java2lab";

private final String DB_USERNAME = "root";

private final String DB_PASSWORD = "";

int rn;

String course;

float marks;

private void connectMeAndFireQuery(){

    try{

        Class.forName(DRIVER_NAME);

        Connection con;

con      =      DriverManager.getConnection(DB_URL,      DB_USERNAME,
DB_PASSWORD);

        String      queryToFire      =      "INSERT      INTO
result(roll_no,course_name,marks_obtained)" + "VALUES(" +this.rn
+", "+"\"'+this.course+"\"'+", "+this.marks+");"

        Statement st;

        st = con.createStatement();

        int n;

        n=st.executeUpdate(queryToFire);

        System.out.println("Thank you "+ n+ " record is added!");

        st.close();

        con.close();

    } catch (Exception e){

```

```

        System.out.println(e);
    }
}

private void getDataFromConsole() {
    Scanner sc = new Scanner(System.in);

    System.out.println("--Enter following informations--");

    System.out.println("---Enter course name--- ");
    course = sc.nextLine();

    System.out.println("---Enter roll number--- ");
    rn = sc.nextInt();

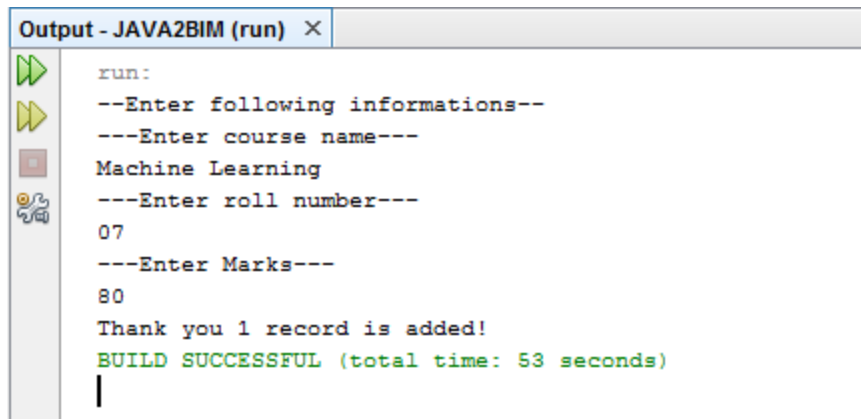
    System.out.println("---Enter Marks--- ");
    marks = sc.nextFloat();
}

public static void main(String[] args) {
    Task1 t1 = new Task1();

    t1.getDataFromConsole();

    t1.connectMeAndFireQuery();
}
}

```



```
run:
--Enter following informations--
---Enter course name---
Machine Learning
---Enter roll number---
07
---Enter Marks---
80
Thank you 1 record is added!
BUILD SUCCESSFUL (total time: 53 seconds)
|
```

```
//Task2 Code

package jdbc_practice;

import java.sql.*;

import java.util.Scanner;

public class Task2 {

    private final String DRIVER_NAME = "com.mysql.jdbc.Driver";

    private          final          String          DB_URL
="jdbc:mysql://localhost:3306/java2lab";

    private final String DB_USERNAME = "root";

    private final String DB_PASSWORD = "";

    int rn;

    String course;

    float marks;

    private void connectMeAndFireQuery(){

        try{

            Class.forName(DRIVER_NAME);

            Connection con;

            con = DriverManager.getConnection(DB_URL, DB_USERNAME,
DB_PASSWORD);

            String queryToFire = "SELECT course_name,marks_obtained"

                + " FROM result"

                +" WHERE roll_no =" +this.rn;

            Statement st;
```



```

    st = con.createStatement();

    ResultSet record;

    record=st.executeQuery(queryToFire);

    while(record.next()){

        course = record.getString("course_name");

        marks = record.getFloat("marks_obtained");

        System.out.println(

            "Roll No: "+rn+"\t "

            +"Course Name: "+ course+ "\t"

            +"Marks: "+marks

            );

    }

    st.close();

    con.close();

} catch (Exception e){

    System.out.println(e);

}

}

private void getDataFromConsole(){

    Scanner sc = new Scanner(System.in);

    System.out.println("---Enter roll number--- ");

    rn = sc.nextInt();

```

```
}

public static void main(String[] args) {

    Task2 t2 =new Task2();

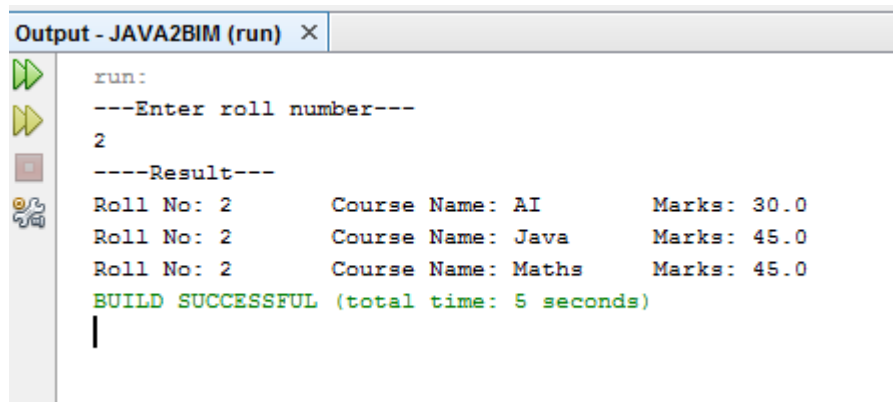
    t2.getDataFromConsole();

    System.out.println("----Result---");

    t2.connectMeAndFireQuery();

}

}
```



```
run:
---Enter roll number---
2
----Result---
Roll No: 2      Course Name: AI      Marks: 30.0
Roll No: 2      Course Name: Java    Marks: 45.0
Roll No: 2      Course Name: Maths   Marks: 45.0
BUILD SUCCESSFUL (total time: 5 seconds)
|
```

## PreparedStatement

Once a connection is obtained we can interact with the database. The JDBC **Statement**, **CallableStatement**, and **PreparedStatement** interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The <b>Statement</b> interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The <b>PreparedStatement</b> interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The <b>CallableStatement</b> interface can also accept runtime input parameters.

- The **PreparedStatement** interface is a subinterface of **Statement**. It is used to execute parameterized query.
- If you want to execute a **Statement** object many times, it usually reduces execution time to use a **PreparedStatement** object instead.
- We should use **PreparedStatement** when we plan to use the SQL statements many times. The **PreparedStatement** interface accepts input parameters at runtime.
- The main feature of a **PreparedStatement** object is that, unlike a **Statement** object, it is given a SQL statement when it is created.
- The usage of **PreparedStatement** is pretty simple. It involves three simple steps as listed below:
  1. Create a **PreparedStatement**
  2. Populate the statement
  3. Execute the prepared statement.

- The basic idea with **PreparedStatement** is using the ‘?’ symbol in the query where ever the data is to be substituted and then when the data is available, replace the symbols with the data.

**Step1:** To create a prepared statement we use the following method on the Connection object as shown below:

```
PreparedStatement prepareStatement (String sql );
```

The SQL query to be passed to the above method will look as shown below:

```
String myQuery = "INSERT INTO students VALUES ( ?, ?, ?, ?, ?)";
```

Since we need to insert data into five columns, we used five question marks. If you notice the query, there are no single quotes like we had in earlier examples. Once we have the above SQL query , we create the prepared statement as shown below:

```
PreparedStatement stmt = con.prepareStatement ( myQuery );
```

**Step 2:** Now that we created a prepared statement, its time to populate it with the data. To do this, we use the **setXXX()** which methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an **SQLException**.

Example:

```
stmt.setInt(1,101);//1 specifies the first parameter in the query
```

```
stmt.setString(2,"Ratan");
```

Always remember one thing. The *numbers in the set methods denote the position of the question mark symbol but not the column number in the table*

If Database Column is	Method to use is
VARCHAR	setString()
NUMERIC or INTEGER	setInt()
DATE OR TIMESTAMP	setDate()

The important methods of `PreparedStatement` interface are given below:

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

Example of **PreparedStatement** that inserts the record

```
package jdbc_practice;

import java.sql.*;

public class PSExample1 {

    final String DRIVER_NAME = "com.mysql.jdbc.Driver";

    final String DB_URL = "jdbc:mysql://localhost:3306/java2lab";

    final String DB_USERNAME = "root";

    final String DB_PASSWORD = "";

    void connectMeAndFireQuery(){

        try{

            Class.forName(DRIVER_NAME);

            Connection con;

            con = DriverManager.getConnection(DB_URL, DB_USERNAME,
            DB_PASSWORD);

            String myQuery = "INSERT INTO students VALUES(?,?,?, ?,?)";

            PreparedStatement pstmt;

            pstmt = con.prepareStatement(myQuery);

            pstmt.setInt(1, 101);

            pstmt.setString(2, "Ratan");

            pstmt.setString(3, "Yadav");

            pstmt.setLong(4, 984100000);

            pstmt.setString(5, "Saptari");
```

```

        int n;

        n=pstmt.executeUpdate();

        System.out.println("Thank you "+ n+ " record is
added!");

        pstmt.close();

        con.close();

    } catch (Exception e){

        System.out.println(e);

    }

}

public static void main(String[] args) {

    PSExample1 pse1 =new PSExample1();

    pse1.connectMeAndFireQuery();

}

}

```

**Example of PreparedStatement interface that retrieve the records of a table**

```

package jdbc_practice;

import java.sql.*;

public class PSExample2 {

    final String DRIVER_NAME = "com.mysql.jdbc.Driver";

    final String DB_URL ="jdbc:mysql://localhost:3306/java2lab";

    final String DB_USERNAME = "root";

```



```

final String DB_PASSWORD = "";

void connectMeAndFireQuery(){

    try{

        Class.forName(DRIVER_NAME);

        Connection con;

        con = DriverManager.getConnection(DB_URL, DB_USERNAME,
DB_PASSWORD);

        String myQuery = "SELECT * FROM result";

        PreparedStatement pstmt;

        pstmt = con.prepareStatement(myQuery);

        ResultSet rs;

        rs=pstmt.executeQuery();

        while(rs.next()){

            int rn = rs.getInt(1);

            String course = rs.getString(2);

            float marks= rs.getFloat(3);

            System.out.println(

                "Roll No: "+rn+"\t "

                +"Course Name: "+ course+ "\t"

                +"Marks: "+marks

            );

        }

        pstmt.close();

```

```
        con.close();  
    } catch (Exception e){  
        System.out.println(e);  
    }  
}  
  
public static void main(String[] args) {  
    PSEExample2 pse2 =new PSEExample2();  
    pse2.connectMeAndFireQuery();  
}  
}
```

Example of **PreparedStatement** interface that deletes the record

```

package jdbc_practice;

import java.sql.*;

import java.util.Scanner;

public class PSExample3 {

    final String DRIVER_NAME = "com.mysql.jdbc.Driver";

    final String DB_URL ="jdbc:mysql://localhost:3306/java21lab";

    final String DB_USERNAME = "root";

    final String DB_PASSWORD = "";

    int rn;

    void getRoll(){

        Scanner sc = new Scanner(System.in);

        System.out.println("Whose data is to be deleted? Enter
Roll No: ");

        rn = sc.nextInt();

    }

    void connectMeAndFireQuery(){

        try{

            Class.forName(DRIVER_NAME);

            Connection con;

            con = DriverManager.getConnection(DB_URL, DB_USERNAME,
DB_PASSWORD);

```

```

String myQuery = "DELETE FROM students WHERE roll_no =? ";

PreparedStatement pstmt;

pstmt = con.prepareStatement(myQuery);

pstmt.setInt(1, rn);

int n = pstmt.executeUpdate();

System.out.println("The record is deleted!");

pstmt.close();

con.close();

} catch (Exception e) {

    System.out.println(e);

}

}

public static void main(String[] args) {

    PSEExample3 pse3 =new PSEExample3();

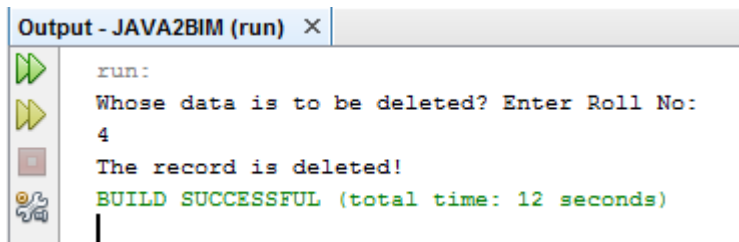
    pse3.getRoll();

    pse3.connectMeAndFireQuery();

}

}

```



```

Output - JAVA2BIM (run) x
run:
Whose data is to be deleted? Enter Roll No:
4
The record is deleted!
BUILD SUCCESSFUL (total time: 12 seconds)

```

**More programming tasks:****Task3:**

**Write a java program using PreparedStatement that allows a user to insert values to a table of particular database (Suppose database is in MySql server ). The program should take the values to insert from console as long**

**Task4:**

**WAP using PreparedStatement to display the records from a table of given database (Suppose database is in MySql server ). Assume the following table :**

**salary(emp\_id , emp\_name ,emp\_salary)**

**The program should read the employee id value from console and display the**

//Task3 code

```
package jdbc_practice;

import java.sql.*;

import java.util.Scanner;

public class Task3 {

    final String DRIVER_NAME = "com.mysql.jdbc.Driver";

    final String DB_URL ="jdbc:mysql://localhost:3306/java2lab";

    final String DB_USERNAME = "root";

    final String DB_PASSWORD = "";

    static int count;

    int rn;

    String course;

    Float marks;

    void getData(){

        Scanner sc1 = new Scanner(System.in);

        System.out.println("Enter Course Name:");

        course =sc1.nextLine();

        System.out.println("Enter Roll Number:");

        rn = sc1.nextInt();

        System.out.println("Enter Marks:");

        marks = sc1.nextFloat();

    }

    void connectMeAndFireQuery(){
```

```

try{

    Class.forName(DRIVER_NAME);

    Connection con;

    con = DriverManager.getConnection(DB_URL, DB_USERNAME,
DB_PASSWORD);

    String myQuery = "INSERT INTO result VALUES(?,?,?)";

    PreparedStatement pstmt;

    pstmt = con.prepareStatement(myQuery);

    pstmt.setInt(1, this.rn);

    pstmt.setString(2,this.course);

    pstmt.setFloat(3,this.marks);

    int n=pstmt.executeUpdate();

    count+=n;

    pstmt.close();

    con.close();

} catch (ClassNotFoundException | SQLException e){

    System.out.println(e);

}

}

public static void main(String[] args) {

    Task3 t3 =new Task3();

    Scanner sc2 = new Scanner(System.in);

    char addMore;

```

```
do{

    t3.getData();

    t3.connectMeAndFireQuery();

    System.out.println("Do    you    want    to    add    more
record? (y/n)");

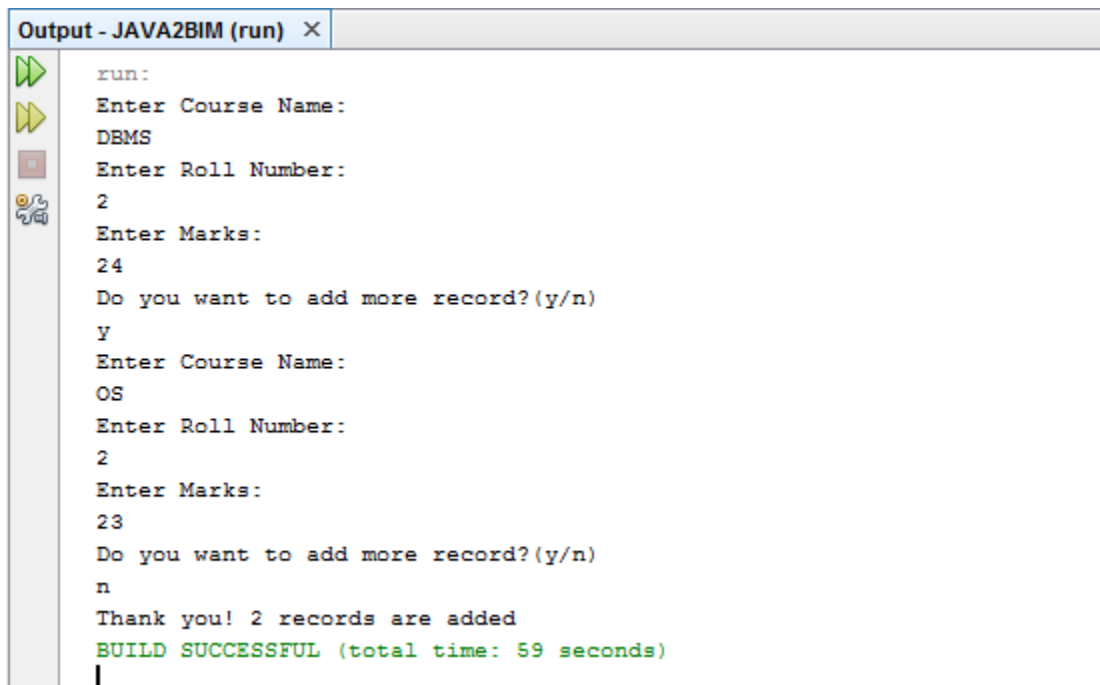
    addMore = sc2.next().charAt(0);

}while(addMore!='n');

System.out.println("Thank you! "+ count +" records are added");

}

}
```



```
Output - JAVA2BIM (run) X
run:
Enter Course Name:
DBMS
Enter Roll Number:
2
Enter Marks:
24
Do you want to add more record?(y/n)
y
Enter Course Name:
OS
Enter Roll Number:
2
Enter Marks:
23
Do you want to add more record?(y/n)
n
Thank you! 2 records are added
BUILD SUCCESSFUL (total time: 59 seconds)
|
```





//Task4 code

```
package jdbc_practice;

import java.sql.*;
import java.util.Scanner;

public class Task4 {

    final String DRIVER_NAME = "com.mysql.jdbc.Driver";

    final String DB_URL ="jdbc:mysql://localhost:3306/java2lab";

    final String DB_USERNAME = "root";

    final String DB_PASSWORD = "";

    int eid;

    void getEmpID(){

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter Employee ID: ");

        eid = sc.nextInt();

    }

    void connectMeAndFireQuery(){

        try{

            Class.forName(DRIVER_NAME);

            Connection con;

            con = DriverManager.getConnection(DB_URL, DB_USERNAME,
DB_PASSWORD);

            String myQuery = "SELECT emp_name,emp_salary FROM salary
WHERE emp_id =?";
```

```

PreparedStatement pstmt;

pstmt = con.prepareStatement(myQuery);

pstmt.setInt(1,this.eid);

ResultSet rs;

rs = pstmt.executeQuery();

while(rs.next())

{

    String name= rs.getString("emp_name");

    double sal = rs.getDouble("emp_salary");

    System.out.println("Employee ID = " +eid +"\t"

        + "Employee Name = "+ name+"\t"

        +"Salary = "+sal

    );

}

pstmt.close();

con.close();

} catch (Exception e){

    System.out.println(e);

}

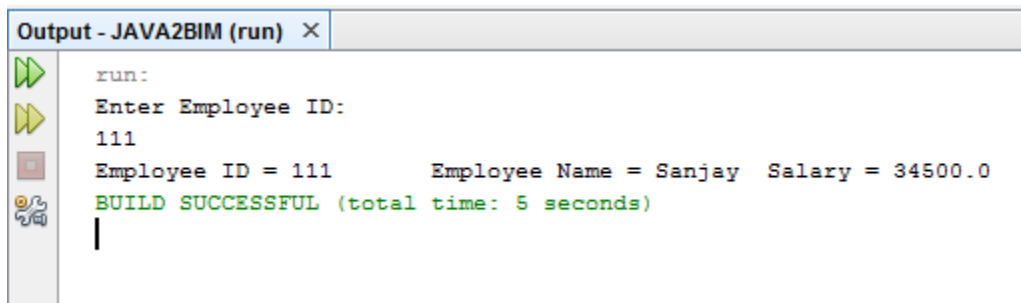
}

public static void main(String[] args) {

    Task4 t4 =new Task4();

```

```
t4.getEmpID();  
  
t4.connectMeAndFireQuery();  
  
}  
  
}
```



```
run:  
Enter Employee ID:  
111  
Employee ID = 111      Employee Name = Sanjay  Salary = 34500.0  
BUILD SUCCESSFUL (total time: 5 seconds)  
|
```

**Query Execution****Example of PreparedStatement interface that inserts the record**

```

package jdbc_practice;

import java.sql.*;

class InsertingPrepared{

public static void main(String args[]){

try{

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

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

PreparedStatement stmt=con.prepareStatement("insert into students values(?,?,?,?)");

stmt.setInt(1, 6);//1 specifies the first parameter in the query

stmt.setString(2,"Ratan");

stmt.setString(3, "Palpa");

stmt.setString(4,"9840456789");

int i=stmt.executeUpdate();

System.out.println(i+" records inserted");

con.close();

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

}

}

```

## **Reading and Writing LOBs**

Many databases can store large objects (LOBs) such as images or other data

### **Types:**

- Binary Large Objects (BLOBs)
- Character Large Objects (CLOBs)

**Reading BLOBs**

```
String sql = "SELECT Cover FROM Covers WHERE
ISBN=?";
PreparedStatement statement =
connection.prepareStatement(sql);
String isbn = "0-201-96426-0";
statement.setString(1, isbn);
ResultSet rs = statement.executeQuery();
if (rs.next()) {
Blob coverBlob = rs.getBlob(1);
Image coverImage =
ImageIO.read(coverBlob.getBinaryStream());
JLabel label = new JLabel(new
ImageIcon(coverImage));
add(label);
}
```

- For **CLOBs**, use **getClob** method of **ResultSet** object

**Example of inserting image in MySQL database**

```
package jdbc_practice;

import java.sql.*;

import java.io.*;

public class InsertingImage {

    public static void main(String[] args) {

        try{

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

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

```

PreparedStatement ps=con.prepareStatement("insert into images values(?,?,?)");

ps.setString(1,"1");

ps.setString(2,"Cat");

FileInputStream fin=new FileInputStream("C:\\Users\\BIPIN\\Desktop\\cat.jpg");

ps.setBinaryStream(3,fin,fin.available());

int i=ps.executeUpdate();

System.out.println(i+" records affected");

con.close();

} catch (Exception e) {e.printStackTrace();}

}

}

```

### Example of retrieving image

```

package jdbc_practice;

import java.sql.*;

import java.io.*;

public class RetrieveImage {

public static void main(String[] args) {

try{

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

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

PreparedStatement ps=con.prepareStatement("select * from images");

ResultSet rs=ps.executeQuery();

if(rs.next()){//now on 1st row

```



```

Blob b=rs.getBlob(2);//2 means 2nd column data
byte barr[]=b.getBytes(1,(int)b.length());//1 means first image

FileOutputStream fout=new
FileOutputStream("C:\\Users\\BIPIN\\Desktop\\cat1.jpg ");
fout.write(barr);

fout.close();
} //end of if
System.out.println("ok");

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

```

### Java Example to store file in database

```

1. import java.io.*;
2. import java.sql.*;
3.
4. public class StoreFile {
5. public static void main(String[] args) {
6. try{
7. Class.forName("oracle.jdbc.driver.OracleDriver");
8. Connection con=DriverManager.getConnection(
9. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
10.
11. PreparedStatement ps=con.prepareStatement(
12. "insert into filetable values(?,?)");
13.
14. File f=new File("d:\\myfile.txt");
15. FileReader fr=new FileReader(f);

```

```

16.
17. ps.setInt(1,101);
18. ps.setCharacterStream(2,fr,(int)f.length());
19. int i=ps.executeUpdate();
20. System.out.println(i+" records affected");
21.
22. con.close();
23.
24. }catch (Exception e) {e.printStackTrace();}
25. }
26. }

```

The example to retrieve the file from the Oracle database is given below.

```

1. import java.io.*;
2. import java.sql.*;
3.
4. public class RetrieveFile {
5.     public static void main(String[] args) {
6.         try{
7.             Class.forName("oracle.jdbc.driver.OracleDriver");
8.             Connection con=DriverManager.getConnection(
9.                 "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
10.
11.             PreparedStatement ps=con.prepareStatement("select * from filetable");
12.             ResultSet rs=ps.executeQuery();
13.             rs.next();//now on 1st row
14.
15.             Clob c=rs.getClob(2);
16.             Reader r=c.getCharacterStream();
17.
18.             FileWriter fw=new FileWriter("d:\\retrivefile.txt");
19.
20.             int i;
21.             while((i=r.read())!=-1)
22.                 fw.write((char)i);
23.

```

```
24. fw.close();  
25. con.close();  
26.  
27. System.out.println("success");  
28. }catch (Exception e) {e.printStackTrace(); }  
29. }  
30. }
```

## SQL Escapes

The "escape" syntax enables common features supported by databases, with database-specific syntax variations, to be written uniformly. JDBC driver to translate the escape syntax to the syntax of a particular database.

### Escapes are provided for the following features

- Date and time literals
- Calling scalar functions
- Calling stored procedures
- Outer joins
- The escape character in LIKE clauses

Example:

- Use d, t, ts for DATE, TIME, or TIMESTAMP values

```
{d '2008-01-24' }
{t '23:59:59' }
{ts '2008-01-24 23:59:59.999' }
```

- Scalar functions

```
{fn left(?, 20) }
{fn user() }
```

- Stored procedures (use = to capture return value)

```
{call PROC1(?, ?) }
{call PROC2}
{call ? = PROC3(?) }
```

- . Outer join

```
SELECT * FROM {oj Books b RIGHT OUTER JOIN
Publishers p ON
b.Publisher_Id = p.Publisher_Id}
```

- Like clause special characters ( % and \_ )

```
... WHERE ? LIKE '%!_%' {escape '!'}
```

### **Processing a Query's ResultSet**

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.

### **Retrieving and Modifying Values from Result Sets**

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.

You access the data in a ResultSet object through a cursor. Note that this cursor is not a database 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**.

### **ResultSet Types**

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.

The sensitivity of a ResultSet object is determined by one of three different ResultSet types:

**(a)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.

**(b)TYPE\_SCROLL\_INSENSITIVE:** The result 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.

**(c)TYPE\_SCROLL\_SENSITIVE:** The result 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.

### **ResultSet Concurrency**

The concurrency of a ResultSet object determines what level of update functionality is supported. There are two concurrency levels:

**CONCUR\_READ\_ONLY:** The ResultSet object cannot be updated using the ResultSet interface.

**CONCUR\_UPDATABLE:** The ResultSet object can be updated using the ResultSet interface.

The default ResultSet concurrency is CONCUR\_READ\_ONLY.

Note: Not all JDBC drivers and databases support concurrency. The method `DatabaseMetaData.supportsResultSetConcurrency` returns true if the specified concurrency level is supported by the driver and false otherwise.

### **Cursor Holdability**

Calling the method `Connection.commit` can close the ResultSet objects that have been created during the current transaction. In some cases, however, this may not be the desired behavior. The

ResultSet property holdability gives the application control over whether ResultSet objects (cursors) are closed when commit is called.

The following ResultSet constants may be supplied to the Connection methods createStatement, prepareStatement, and prepareCall:

**HOLD\_CURSORS\_OVER\_COMMIT:** ResultSet cursors are not closed; they are holdable: they are held open when the method commit is called. Holdable cursors might be ideal if your application uses mostly read-only ResultSet objects.

**CLOSE\_CURSORS\_AT\_COMMIT:** ResultSet objects (cursors) are closed when the commit method is called. Closing cursors when this method is called can result in better performance for some applications.

The default cursor holdability varies depending on your DBMS.

### **Retrieving Column Values from Rows**

The ResultSet interface declares **getter methods** (for example, **getBoolean** and **getLong**) for retrieving column values from the current row. You can retrieve values using either the index number of the column or the alias or name of the column. The column index is usually more efficient. Columns are numbered from 1. For maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.

```
try
{

    // create Statement for querying database
    statement = connection.createStatement();

    // query database
    resultSet = statement.executeQuery(
        "SELECT AuthorID, FirstName, LastName FROM authors" );
```

```

// process query results
ResultSetMetaData metaData = resultSet.getMetaData();
int numberOfColumns = metaData.getColumnCount();
System.out.println( "Authors Table of Books Database:\n" );

for ( int i = 1; i <= numberOfColumns; i++ )
    System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
System.out.println();

while ( resultSet.next() )
{
    int id = rs.getInt("AuthorID");
    String firstName = rs.getString("FirstName");
    String lastName = rs.getString("LastName");
    System.out.println(id+ "\t" + firstName+
        "\t" + lastName );
} // end while
} // end try
catch ( SQLException sqlException )
{
    sqlException.printStackTrace();
} // end catch

```

### **Cursors**

As mentioned previously, you access the data in a **ResultSet object through a cursor**, which points to one row in the ResultSet object. However, when a ResultSet object is first created, the cursor is positioned before the first row. There are other methods available to move the cursor:

**next:** Moves the cursor forward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.

**previous:** Moves the cursor backward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned before the first row.



**first:** Moves the cursor to the first row in the ResultSet object. Returns true if the cursor is now positioned on the first row and false if the ResultSet object does not contain any rows.

**last:** Moves the cursor to the last row in the ResultSet object. Returns true if the cursor is now positioned on the last row and false if the ResultSet object does not contain any rows.

**beforeFirst:** Positions the cursor at the start of the ResultSet object, before the first row. If the ResultSet object does not contain any rows, this method has no effect.

**afterLast:** Positions the cursor at the end of the ResultSet object, after the last row. If the ResultSet object does not contain any rows, this method has no effect.

**relative(int rows):** Moves the cursor relative to its current position.

**absolute(int row):** Positions the cursor on the row specified by the parameter row.

Note that the default sensitivity of a ResultSet is TYPE\_FORWARD\_ONLY, which means that it cannot be scrolled; you cannot call any of these methods that move the cursor, except next, if your ResultSet cannot be scrolled.

### Updating Rows in ResultSet Objects

You cannot update a default ResultSet object, and you can only move its cursor forward. However, you can create ResultSet objects that can be scrolled (the cursor can move backwards or move to an absolute position) and updated.

```
try {
    // establish connection to database
    connection = DriverManager.getConnection(
        DATABASE_URL, "root", "" );
    statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
    ResultSet uprs= statement.executeQuery(
        "SELECT * FROM authors");

    while (uprs.next()) {
        uprs.updateString( "LastName","Sharma");
    }
}
```

```

        uprs.updateRow();
    }
}
catch ( SQLException sqlException )
{
    sqlException.printStackTrace();
} // end catch

```

The field **ResultSet.TYPE\_SCROLL\_SENSITIVE** creates a ResultSet object whose cursor can move both forward and backward relative to the current position and to an absolute position. The field **ResultSet.CONCUR\_UPDATABLE** creates a ResultSet object that can be updated. See the ResultSet Javadoc for other fields you can specify to modify the behavior of ResultSet objects.

The method **ResultSet.updateString** updates the specified column (in this example, LastName with the specified float value in the row where the cursor is positioned. ResultSet contains various updater methods that enable you to update column values of various data types. However, none of these updater methods modifies the database; you must call the method **ResultSet.updateRow to update the database.**

### **Inserting Rows in ResultSet Objects**

Note: Not all JDBC drivers support inserting new rows with the ResultSet interface. If you attempt to insert a new row and your JDBC driver database does not support this feature, a `SQLFeatureNotSupportedException` exception is thrown.

```

try {
    statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);

    ResultSet uprs = statement.executeQuery(
        "SELECT * FROM authors");

    uprs.moveToInsertRow();

```

```

        uprs.updateInt("AuthorID",9);
        uprs.updateString("FirstName","Subash");
        uprs.updateString("LastName","Pakhrin");
        uprs.insertRow();
        uprs.beforeFirst();
    }
    catch ( SQLException sqlException )
    {
        sqlException.printStackTrace();
    } // end catch

```

This example calls the `Connection.createStatement` method with two arguments, **`ResultSet.TYPE_SCROLL_SENSITIVE`** and **`ResultSet.CONCUR_UPDATABLE`**. The first value enables the cursor of the `ResultSet` object to be moved both forward and backward. The second value, `ResultSet.CONCUR_UPDATABLE`, is required if you want to insert rows into a `ResultSet` object; it specifies that it can be updatable.

The same stipulations for using strings in getter methods also apply to updater methods.

The method **`ResultSet.moveToInsertRow`** moves the cursor to the insert row. The insert row is a special row associated with an updatable result set. It is essentially a **buffer** where a new row can be constructed by calling the updater methods prior to inserting the row into the result set. For example, this method calls the method `ResultSet.updateString` to update the insert row's `COF_NAME` column to Kona.

The method `ResultSet.insertRow` inserts the contents of the insert row into the `ResultSet` object and into the database.

Note: After inserting a row with the `ResultSet.insertRow`, you should move the cursor to a row other than the insert row. For example, this example moves it to before the first row in the result set with the method **`ResultSet.beforeFirst`**. Unexpected results can occur if another part of your

application uses the same result set and the cursor is still pointing to the insert row.

### **Using Statement Objects for Batch Updates**

Statement, PreparedStatement and CallableStatement objects have a list of commands that is associated with them. This list may contain statements for updating, inserting, or deleting a row; and it may also contain DDL statements such as CREATE TABLE and DROP TABLE. It cannot, however, contain a statement that would produce a ResultSet object, such as a SELECT statement. In other words, the list can contain only statements that produce an update count.

The list, which is associated with a Statement object at its creation, is initially empty. You can add SQL commands to this list with the method addBatch and empty it with the method clearBatch. When you have finished adding statements to the list, call the method **executeBatch** to send them all to the database to be executed as a unit, or batch.

```
try {
    connection = DriverManager.getConnection(
        DATABASE_URL, "root", "" );
    connection.setAutoCommit(false);
    statement = connection.createStatement();

    statement.addBatch(
        "INSERT INTO authors " +
        "VALUES('15','Hari','Shrestha')");

    statement.addBatch(
        "INSERT INTO authors " +
        "VALUES('16','Ram','Acharya')");

    statement.addBatch(
        "INSERT INTO authors " +
        "VALUES('17','Shyam','Gautam')");
```

```

statement.addBatch(
    "INSERT INTO authors " +
    "VALUES('18','Govinda','Paudel')");

int [] updateCounts = statement.executeBatch();
connection.commit();

} catch (BatchUpdateException b) {
    b.printStackTrace();
} catch (SQLException ex) {
    ex.printStackTrace();
}

```

The following line disables auto-commit mode for the Connection object con so that the **transaction** will not be automatically committed or rolled back when the method executeBatch is called.

```
connection.setAutoCommit(false);
```

To allow for correct error handling, you should always disable auto-commit mode before beginning a batch update.

The method **Statement.addBatch** adds a command to the list of commands associated with the Statement object statement. In this example, these commands are all INSERT INTO statements, each one adding a row consisting of three column values.

The following line sends the four SQL commands that were added to its list of commands to the database to be executed as a batch:

```
int [] updateCounts = statement.executeBatch();
```

Note that **statement** uses the method **executeBatch** to send the batch of insertions, **not the method executeUpdate, which sends only one command and returns a single update count.** The

DBMS executes the commands in the order in which they were added to the list of commands, so it will first add the row of values for "Hari" , then add the row for "Ram", then "Shyam" , and finally "Govinda". If all four commands execute successfully, the DBMS will return an update count for each command in the order in which it was executed. The update counts that indicate how many rows were affected by each command are stored in the array `updateCounts`.

**If all four of the commands in the batch are executed successfully, `updateCounts` will contain four values, all of which are 1 because an insertion affects one row.** The list of commands associated with `stmt` will now be empty because the four commands added previously were sent to the database when `stmt` called the method `executeBatch`. You can at any time explicitly empty this list of commands with the method `clearBatch`.

**The `Connection.commit` method makes the batch of updates to the "authors" table permanent. This method needs to be called explicitly because the auto-commit mode for this connection was disabled previously.**

The following line enables auto-commit mode for the current `Connection` object.

```
connection.setAutoCommit(true);
```

Now each statement in the example will automatically be committed after it is executed, and it no longer needs to invoke the method `commit`.

## Multiple Results

A query can return multiple results (when executing a stored procedure, or in databases that allow multiple `SELECT` statements in a single query).

### Steps to process multiple results

- Use the `execute` method to execute the SQL statement
- Retrieve the first result or update count

- Call *getMoreResults* method to move on to the next result set
- Exit when there are no more result sets and no update counts
- Example:

```
String url =
"jdbc:sqlserver://localhost\\SQLEXPRESS;"
+
"databaseName=Demo;integratedSecurity=true";
Connection connection =
DriverManager.getConnection(url);
Statement statement =
connection.createStatement();
String sql = "select * from Publishers;"
+ "insert into Publishers values ("
+ "'1002', 'Test_Publisher2',
'www.example.com');"
boolean isResultSet =
statement.execute(sql);

while (true) {
if (isResultSet) { // SELECT query
ResultSet rs = statement.getResultSet();
while(rs.next()) {
System.out.printf("%-30s %-15s \n",
rs.getString("Name"),
rs.getString("Url"));
}
} else { // other queries
int updateCount = statement.getUpdateCount();
if (updateCount >= 0) {
System.out.println("Record inserted
successfully.");
} else { // no more results
```

```
break;
}
}
    isResultSet = statement.getMoreResults();

}
```

### **Scrollable Result Sets**

#### **Obtaining scrollable record set**

```
Statement s = connection.createStatement(
type, concurrency);
```

**OR**

```
PreparedStatement s =
connection.prepareStatement(sql, type,
concurrency);
```



**ResultSet Type Values**❖ `TYPE_FORWARD_ONLY`

The result set is not scrollable (default)

❖ `TYPE_SCROLL_INSENSITIVE`

The result set is scrollable but not sensitive to database changes

❖ `TYPE_SCROLL_SENSITIVE`

The result set is scrollable and sensitive to database changes

**ResultSet Concurrency Values**❖ `CONCUR_READ_ONLY`

The result set cannot be used to update the database (default)

❖ `CONCUR_UPDATABLE`

The result set can be used to update the database

**Updatable Result Sets**

Changes in the result set are automatically reflected in the database. Updatable result sets don't have to be scrollable (but they usually are).

**Examples****Update existing record**

```
Statement statement =
connection.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE
);
String sql = "select * from Authors";
ResultSet rs = statement.executeQuery(sql);
if (rs.next()) {
String name = rs.getString("Name");
```

```
rs.updateString("Name", name + "
(Modified)");
rs.updateRow();
}
```

❖ Note : It is more efficient to use UPDATE statements than updatable result sets.

### **Insert new record**

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

### **Delete current record**

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

### **Row Sets**

Scrollable result sets need to keep the database connection open during the entire user interaction (not a good idea as database connections are expensive)

The RowSet (extends ResultSet interface) is not tied to a database connection

#### Interfaces that extend RowSet

- CachedRowSet  
allows disconnected operation
- WebRowSet  
cached row set that can be saved to an XML file

- **FilteredRowSet** and **JoinRowSet**  
support lightweight operations on row sets that are equivalent to SQL SELECT and JOIN operations
- **JdbcRowSet**
  - thin wrapper around a **ResultSet**
  - adds useful getters and setters from the **RowSet** interface, turning a result set into a "bean."

### Cached Row Sets

**CachedRowSet** is a subinterface of the **ResultSet** interface. A cached row set caches all data from a result set in local memory. It can be used/modified even after closing the database connection. Modifications can later be propagated to database

### Examples

```
ResultSet rs = statement.executeQuery(sql);
CachedRowSet crs = new CachedRowSetImpl();
// or use an implementation from your database
// vendor
```

```
crs.populate(rs);
connection.close();
while(crs.next()) { ... }
```

#### *Alternatively,*

```
CachedRowSet crs = new CachedRowSetImpl();
crs.setUrl(url);
crs.setUsername(username);
crs.setPassword(password);
crs.setCommand(sql);
crs.execute(); // connect, execute sql, populate
row set and disconnect
while(crs.next()) { ... }
```

- Update database with modification in cached row set

```
crs.acceptChanges(connection);
OR
crs.acceptChanges();
```

Note : *The second call works only if you configured the row set with connection information (such as URL, user name, and password).*

*CachedRowSetImpl updates database records only if they have not changed by other users (between retrieval and update by current user); otherwise, a SyncProviderException is thrown and update fails.*

**If your query result is very large, use paging to retrieve a limited number of rows at a time**

```
CachedRowSet crs = . . . ;
crs.setCommand(sql);
crs.setPageSize(20);
. . .
crs.execute();
```

- Retrieve the next batch of rows (page) with  

```
boolean hasNextPage = crs.nextPage();
```

## Transactions

You can group a set of statements to form a transaction. The transaction can be committed when all statements are executed without any problem. If an error has occurred in one of them, all completed statements can be rolled back as if none of the statements had been issued. Usage scenario: In balance transfer from an account to another, one account should be debited and another credited. Either both of them should succeed or both of them should fail.

## Transaction Processing

Many database applications require guarantees that a series of database insertions, updates and deletions executes properly before the application continues processing the next database operation. For example, when you transfer money electronically between bank accounts, several

factors determine if the transaction is successful. You begin by specifying the source account and the amount you wish to transfer from that account to a destination account. Next, you specify the destination account. The bank checks the source account to determine whether its funds are sufficient to complete the transfer. If so, the bank withdraws the specified amount and, if all goes well, deposits it into the destination account to complete the transfer. What happens if the transfer fails after the bank withdraws the money from the source account? In a proper banking system, the bank redeposits the money in the source account. The way to be sure that either both actions occur or neither action occurs is to use a **transaction**. **A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements is executed.**

The way to allow two or more statements to be **grouped into a transaction** is to **disable the auto-commit mode**.

### **Disabling Auto-Commit Mode**

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.

The way to allow two or more statements to be **grouped into a transaction** is to **disable the auto-commit mode**.

```
con.setAutoCommit(false);
```

### **Committing Transactions**

After the auto-commit mode is disabled, no SQL statements are committed until you call the **method commit explicitly**. All statements executed after the previous call to the method commit are included in the **current transaction and committed together as a unit**.

```
con.commit();
```

### **Rollback**

If you group update statements to a transaction, then the transaction either succeeds in its entirety and it can be committed, or it fails somewhere in the middle. In that case, you can carry out a **rollback** and the database automatically undoes the effect of all updates that occurred since the last committed transaction.

You turn off autocommit mode with the command

```
conn.setAutoCommit(false);
```

Now you create a statement object in the normal way:

```
Statement stat = conn.createStatement();
```

Call executeUpdate any number of times:

```
stat.executeUpdate(command1);
```

```
stat.executeUpdate(command2);
```

```
stat.executeUpdate(command3);
```

```
...
```

When all commands have been executed, call the commit method:

```
conn.commit();
```

However, if an error occurred, call

```
conn.rollback();
```

Then, all commands until the last commit are automatically reversed. You typically issue a rollback when your transaction was interrupted by a SQLException.

### **Save Points**

You can gain finer-grained control over the rollback process by using save points. Creating a save point marks a point to which you can later return without having to return to the start of the transaction. For example,

```
Statement stat = conn.createStatement();    // start transaction; rollback() goes here
```

```
stat.executeUpdate(command1);
```

```
Savepoint svpt = conn.setSavepoint();        // set savepoint; rollback(svpt) goes here
```

```
stat.executeUpdate(command2);
```

```
if (. . .) conn.rollback(svpt);                // undo effect of command2
```

```
...
```

```
conn.commit();
```

Here, we used an anonymous save point. You can also give the save point a name, such as

```
Savepoint svpt = conn.setSavepoint("stage1");
```

When you are done with a save point, **you should release it:**

```
stat.releaseSavepoint(svpt);
```

**Example:**

```
connection.setAutoCommit(false); // disable auto
commit after each "execute"
Statement statement = connection.createStatement();
String debitSql = "Update Accounts set balance =
balance - 100 "
+ "where Account_Id = 1";
String creditSql = "Update Accounts set balance =
balance + 100 "
+ "where Account_Id = 2";
try {
statement.executeUpdate(debitSql);
statement.executeUpdate(creditSql);
connection.commit();
System.out.println("Balance transferred!");
}
catch(SQLException e) {
connection.rollback();
System.out.println("Transfer failed");
}
finally {
connection.close();
}
```

**Example of transaction processing**

```

1. import java.sql.*;
2. import java.io.*;
3. class TM{
4. public static void main(String args[]){
5. try{
6.
7. Class.forName("oracle.jdbc.driver.OracleDriver");
8. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9. con.setAutoCommit(false);
10.
11. PreparedStatement ps=con.prepareStatement("insert into user420 values(?,?,?)");
12.
13. BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
14. while(true){
15.
16. System.out.println("enter id");
17. String s1=br.readLine();
18. int id=Integer.parseInt(s1);
19.
20. System.out.println("enter name");
21. String name=br.readLine();
22.
23. System.out.println("enter salary");
24. String s3=br.readLine();
25. int salary=Integer.parseInt(s3);
26.
27. ps.setInt(1,id);
28. ps.setString(2,name);
29. ps.setInt(3,salary);
30. ps.executeUpdate();
31.
32. System.out.println("commit/rollback");
33. String answer=br.readLine();

```



```
34. if(answer.equals("commit")){
35. con.commit();
36. }
37. if(answer.equals("rollback")){
38. con.rollback();
39. }
40.
41.
42. System.out.println("Want to add more records y/n");
43. String ans=br.readLine();
44. if(ans.equals("n")){
45. break;
46. }
47.
48. }
49. con.commit();
50. System.out.println("record successfully saved");
51.
52. con.close();//before closing connection commit() is called
53. }catch(Exception e){System.out.println(e);}
54.
55. }}
```

## References

1. [javatpoint.com](http://javatpoint.com)
2. [tutorialspoint.com](http://tutorialspoint.com)
3. [studytonight.com](http://studytonight.com)
4. [way2java.com](http://way2java.com)
5. “Advanced Java Programming “ by *Arjun Sing Saud*
6. “Core Java Volume II – Advanced Features, Seventh Edition “ by Cay S. Horstman, Gray Cornell