In This Chapter

CHAPTER 8

Database Connectivity to MySQL

8.1 Introduction

When you design real-life applications, you are bound to encounter situations wherein you need to manipulate data stored in a database through an application designed by you. Since you are developing Java GUI applications using NetBeans IDE, in this chapter, our discussion will remain limited to how to connect to database in a Java based application.

In order to connect to a database from within an application, you need a framework that helps you send and execute SQL statements from within the application code. The most popular programming interface for accessing relational database is ODBC (Open DataBase Connectivity). But ODBC is Microsoft's API which cannot be directly used from within Java. If you want to connect to a relational database (*e.g.*, MySQL's or Oracle's) from within a Java application, you can do it using JDBC (Java DataBase Connectivity) API of Java. You may even use JDBC-ODBC connector that is JDBC layer over ODBC but JDBC is a better choice for accessing databases like MySQL.

NOTE

If you need to access a database which is traditionally accessed via ODBC from Java (e.g., MS Access) then a special type of driver is needed which acts as a bridge between JDBC and ODBC. It is called jdbc.odbc bridge.

What Does JDBC Do ?

JDBC mainly does following tasks from within a Java application code :

(*i*) establish a connection with a database.

(*ii*) send SQL statements to database server.

(*iii*) process the results obtained.

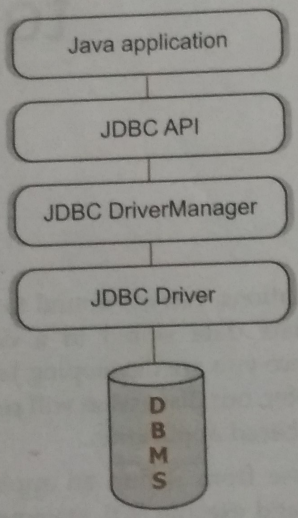
In this chapter, you are going to learn how you can establish database connection from your Java applications and process the retrieved data.

8.2 Classes used for Database Connectivity

The JDBC API consists of a set of Java classes that use predefined methods to handle various data access functions such as *selecting the appropriate database driver to access a particular database, connecting to the database, submitting SQL statements to the DBMS,* and *processing the results returned by the DBMS.*

There are *four* main classes in the JDBC API hierarchy that are generally used for database connectivity. These are :

1. **DriverManager Class**. The JDBC *DriverManager* class loads the *JDBC driver* needed to access a particular data source, locates and logs on to the database and returns a *Connection* object.
2. **Connection Class**. The JDBC *Connection* class manages the communications between a *Java client application* and a specific database (*e.g.* MySQL database), including passing SQL statements to the DBMS and managing transactions.
3. **Statement Class.** The JDBC *Statement* class contains SQL strings that are submitted to the DBMS. An *SQL Select statement* returns a *ResultSet* object that contains the data retreived as the result of SQL statement.
4. **ResultSet Class.** The JDBC *ResultSet* class provides predefined methods to access, analyze, and convert data values returned by an executed *SQL Select statement*.



JDBC API handles communication between its upper layer (i.e., Java application) and its lower layer(i.e., the DriverManager)

JDBC Driver API handles communication between its upper layer (i.e., Driver Manger) and its lower layer (i.e., the actual driver)

Figure 8.1 JDBC Architecture.

The required JDBC driver must be registered with the JDBC Driver Manager before a connection between the Java application and the JDBC data source can be made. You'll learn later that we do it using Class.forName( ) method. Once a driver is registered and loaded, the *DriverManager* can use that driver to connect the Java application to its associated data source *i.e.,* dbms. Once the connection is established, all database calls go directly to the JDBC driver itself, bypassing the *DriverManager*.

8.3 Prerequisites for Connecting to MySQL from JAVA

When you connect *two* different types of application *e.g.,* Java (an *OO programming language*) and MySQL (an *RDBMS*), you need a software that can act as a bridge between the two. To establish connection between Java and MySQL, you need a software that is called **MySQL Connector/J**.

MySQL describes this software as

> "MySQL provides connectivity for client applications developed in the Java programming language via a JDBC driver, which is called MySQL Connector/J".

Before you go on to write database connectivity program, make sure that this driver is available and installed on your machines.

You can download it from URL :

```
dev.mysql.com/downloads/
```

You can determine whether MySQL connector is available with your NetBeans IDE or not by following the steps given below :

(*i*) Start NetBeans IDE.

(*ii*) Click **Tools → Libraries.**

(*iii*) Now, the dialog **Library Manager** will open up. Here, in the **Libraries box**, look for **MySQL JDBC Driver** under *class Libraries*. If it is there, that means this driver is already installed on your machine (see Fig. 8.2). Note down its path (*i.e.*, location on your machine) so that when you develop your database-connectivity application, you can locate it from this very folder.

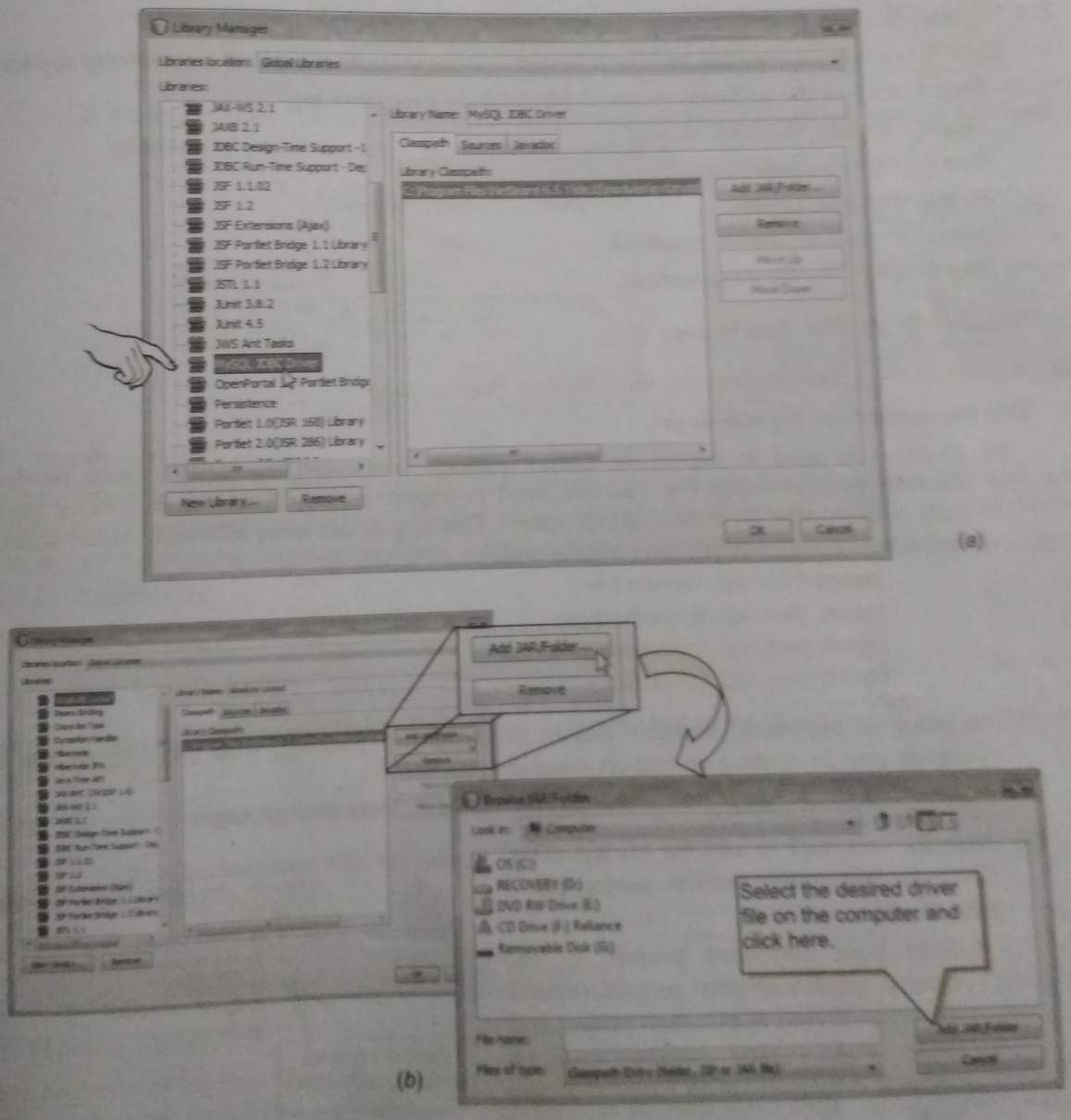


Figure 8.2 Adding MySQL JDBC Driver to NetBeans IDE.

This step would be covered in section 8.4.1.

- If you do not find the JDBC driver there, then you need to first download it from the URL mentioned above *i.e.*, from *dev.mysql.com/downloads/* and extract it from its compressed form into a folder.
- After this, you need to follow first two steps as described above.
- In the **Library Manager** dialog, you need to click at **Add JAR/Folder...** button (see Fig. 8.2) and then select the downloaded and extracted driver file and click **Add JAR/Folder** button.

8.4 Connecting to MySQL from JAVA

After the prerequisites of connectivity from Java to MySQL, let us now learn to create applications that perform database connectivity.

8.4.1 Steps for Creating Database Connectivity Applications

There are mainly *six* steps that must be followed in order to create a database connectivity application.

Step 1 Import the packages required for database programming.

Step 2 Register the JDBC driver.

Step 3 Open a connection.

Step 4 Execute a query.

Step 5 Extract data from result set.

Step 6 Clean up the environment.

These are being described below :

Step 1 : *Import the Packages Required for Database Programming*

This step consists of *two* sub-steps :

(*a*) *Firstly*, you need to import the library package containing the JDBC classes needed for database programming. For this, you need to import various classes from java.sql package, such as *Connection, DriverManager, Statement*. That is, you can write following *import* statements at the top of your application's source code :

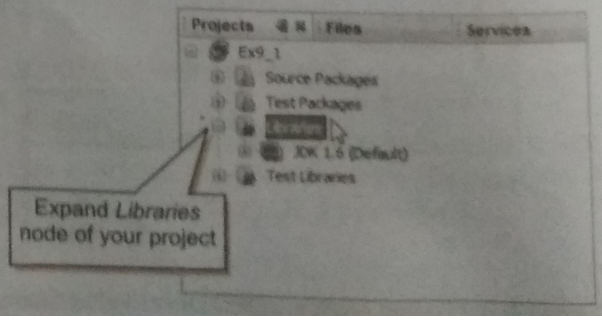
```
import java.sql.Connection ;
import java.sql.DriverManager ;
import java.sql.Statement ;
import java.sql.ResultSet ;
```

In place of importing separate classes from java.sql package, you can import the entire package also by writing an import command as follows :

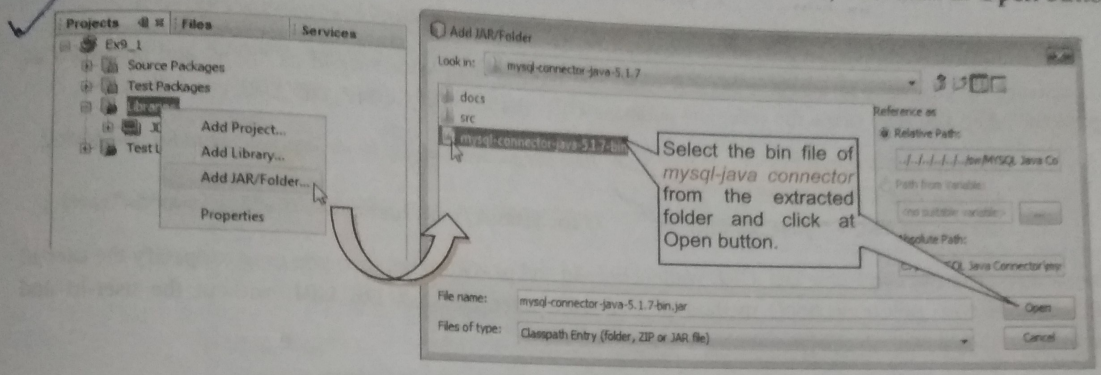
```
import java.sql.* ;          //STEP 1. Import required packages
```

(*b*) *Secondly*, you need to add the *MySQL JDBC Connector* that is discussed in section 8.2 to your NetBeans project.

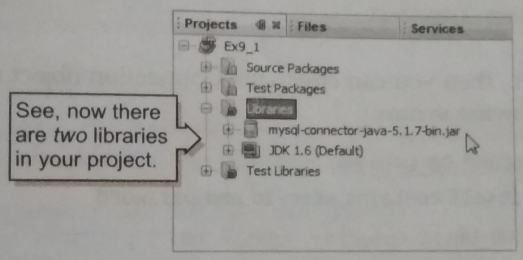
- Start your NetBeans project. In the *Projects window* of your project, expand the node of your application project and under that, expand *Libraries node* by clicking the plus icon in front of it. (*see* here)



- If you don't see *mysql-connector* here, then you need to add *MySQL Connector* under the *Library* node. For this, right click on the **Libraries** node and click at **Add JAR/Folder...** option (see below). Then browse to go to the folder where extracted copy of this *MySQL Connector* lies. Open the folder, select the *bin file* (see below) and click at **Open** button.



- Now you'll see *two* libraries beneath the *Libraries* node (see below) — one the *JDK* and another the *mysql-connector* that you added just now.



NOTE

Please note, it is important to add the required packages/ libraries to your project, otherwise you won't be able to connect to database successfully from within your application.

Step 2 : Register the JDBC Driver

In this step, you need to initialise a driver so you can open a *communications channel* with the database from within the Java application. For this, you need to first register the jdbc driver with the *DriverManager*. The *java.lang* package's one class namely **Class** offers a method called **forName( )** that registers the driver with the *DriverManager*.

Following is the synatx to achieve this :

```
Class.forName("<driver's name>") ;
```

For example, you can connect to MySQL using one of the following *two* drivers : *java.sql.Driver* or *com.mysql.jdbc.Driver*.

Thus, you need to write one of the following *two* statements :

```
//STEP 2 : Register JDBC driver
    Class.forName("java.sql.Driver") ;
```

You can use any of these two JDBC drivers to connect to MySQL database from within a Java application.

Or

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

Step 3 : Open a Connection

This requires using the *DriverManager.getConnection( )* method to create a *Connection* object, which represents a physical connection with the database. *The* ***Connection*** *object allows us to establish a physical connection to the data source (RDBMS here).*

To connect to a database, you need to know *database's complete URL*, the *user's id* and *password*. You must be familiar with user-id and password for accessing your MySQL database.

Let us learn how you can frame the URL of your database. A MySQL database's URL is framed as per following syntax :

```
jdbc:mysql://localhost/<database-name>? user= "username" &password="password"
```

For instance, if you want to access a MySQL database namely *test*, with *user-id* as "root" and *password* as "goodone", then you may frame the complete database-URL (let's call it COMP_DB_URL) as follows:

```
String uid = "root" ;
String pwd = "goodone" ;
String DB_URL = "jdbc:mysql://localhost/test?user="+uid+"&password="+pwd ;
```

Here the password should be the one, you use to log into MySQL.

Alternatively, you can also frame DB_URL without user-id and password but then you need to specify the user-id and password with *getConnection()* method of *DriverManager* class. A DB_URL without the user-id and password is framed as :

```
jdbc:mysql://localhost[:<portnumber>]/<database-name>
```

Mostly the port number 3306 is used (the default port), so your DB_URL will become :

```
jdbc:mysql://localhost:3306/test
```

Once you have framed the DB_URL, then you can create the Connection object using *DriverManager* class's *getConnection* method as per following syntax :

```
DriverManager.getConnection(<COMP_DB_URL>)
                    if the url itself contains user-id and password
Or  DriverManager.getConnection(<DB_URL>, <userid>, <password> ) ;
                    if the DB_URL does not contain user-id and password
```

The *getConnection()* method returns a *Connection* object, thus you must store its return value in a *Connection object*. That means, complete code to open a connection will be :

```
String UID = "username" ;
String PWD = "password" ;
String DB_URL= "jdbc:mysql://localhost:3306/test" ;
Connection conn = DriverManager.getConnection(DB_URL, UID, PWD) ;
```

Or

* if you have complete database URL (*i.e.*, COMP_DB_URL) then write :

```
String COMP_DB_URL=" jdbc:mysql://localhost/test?user="+UID+"&password="+PWD ;
Connection conn = DriverManager.getConnection(COMP_DB_URL) ;
```

If you feel that framing DB_URL is a bit confusing, then you can directly read the DB_URL from your NetBeans IDE by following the steps given below :

(*i*) Open **Services window** of your NetBeans project by pressing **Ctrl+5**.

(*ii*) Expand *Databases* node by clicking its plus icon.

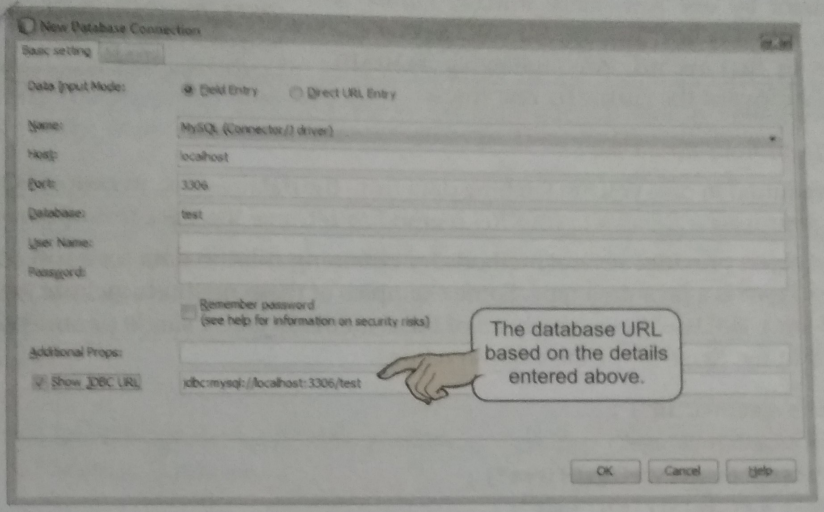
(*iii*) Right click on **Databases** node and click at **New Connection...**

DEF

A **Connection** (represented through Connection object) is the session between the application program and the database. To do anything with database, one must have a connection object.

(iv) In the **New Database Connection** dialog, select **Name :** as MySQL(Connector/J driver) ; specify **Host :** as localhost ; **Port :** as 3306 and **Database :** as your database name.

(v) Then check the box at the bottom : **Show JDBC URL**, and *voila..*, we get the DB_URL we wanted.



Step 4 : Execute a Query

In this step, you need to first create an object of type Statement or PreparedStatement for building and submitting an SQL statement to the database using createStatement( ) method of *Connection type object*.

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

In the above code, conn is the same *Connection* object that you created in step 3.

Next you need to execute the SQL statement using executeQuery( ) method as explained below. The executeQuery( ) method returns a resultset (an object of *ResultSet* type) that contains the resultant dataset of the executed query. This, you must store the returned value of *executeQuery*( ) into a *ResultSet* object.

NOTE

*The **Statement object** is the object used for executing one static SQL statement on the associated database-table and returning the results it produces.*

```
String sql ;
sql = "SELECT id, first, last, age FROM Employees ; " ;
ResultSet rs = stmt.executeQuery(sql) ;
```

Add a semicolon in SQL query and then end your java statement with a semicolon.

In the above code, stmt is the same *Statement* object that you created just before.

If, however, the SQL statement is an UPDATE, INSERT or DELETE statement then you can execute SQL query using ***executeUpdate***[1]( ) method as shown below :

```
Statement stmt = conn.createStatement() ;
String sql ;
sql = "DELETE FROM Employees" ;
ResultSet rs = stmt.executeUpdate(sql) ;
```

DEF

A **result set** (represented by a ResultSet object) refers to a logical set of records that are fetched from the database by executing a query and made available to the application-program.

1. There are some other SQL statements (*e.g.*, Create Table) also that can be executed with executeUpdate() method.

ResultSet Cursor

When a ResultSet object is first created, **the cursor is positioned before the first row**. To move the cursor to first row, you can either write **rs.next()** or **rs.first()**. **rs.next()** forwards the cursor by one row- since initially cursor is before the first row, first **rs.next()** command would place the result-set cursor to first record. Any following **rs.next()** commands would forward the cursor by one row.

NOTE

If the SQL query is a SELECT query then it is to be executed using method

<statement-object>.executeQuery(<query>),

but if the SQL query is INSERT/DELETE/UPDATE query then it is to be executed using method

<statement-object>.executeUpdate(<query>)

Step 5 : *Extract Data from Result Set*

This step is required in case you are fetching data from the database *i.e.*, in case of SQL SELECT query. You can use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.

The *ResultSet* object provides several methods for obtaining column data for a row. All are of the form *get<Type>*, where *<Type>* is a Java data type. Some examples of these methods include getInt( ), getLong( ), getString( ), getFloat( ), getDate( ) etc. Nearly all of these methods take a single parameter that is either the column index within the *ResultSet* or the column name.

```
int id = rs.getInt("id") ;
int age = rs.getInt("age") ;
String first = rs.getString("first") ;
String last = rs.getString("last") ;
```

If the employee table's structure is like : *id, firstname, lastname, age*, then you can either retrieve the data as shown in above code or through following code that uses the order of the columns :

```
int id  = rs.getInt(1) ;              //retrieves the 1st column (int type)
String first = rs.getString(2) ;      //retrieves the 2nd column (String type)
String last = rs.getString(3) ;       //retrieves the 3rd column (String type)
int age = rs.getInt(4) ;              //retrieves the 4th column (int type)
```

ResultSet columns are numbered, starting with 1. If the column name is used and if there is more than one column in the ResultSet with the same name, the first one is returned.

Now you can use the retrieved values in your application the way you want.

Hey stop! You did not notice one thing. All the above statements are going to fetch data from different columns of one row only. What if the SQL query returned multiple rows? That is, if the result-set contains multiple rows, then how would you retrieve data from all the rows of the record-set ? Good, you guessed it right — *through a loop*. But can you answer this— till when should the loop repeat ? Simple! As long as there are rows left in the resultset and this is determined through *next*( ) method of resultset.

The ***next()* method moves the cursor forward in a result-set by one row.** It returns *true* if the cursor is now positioned on a row and *false* if the cursor is positioned after the last row. That means, to process the complete result-set, your code should be like :

```
int id, age ;
String first, last ;
while (rs.next() ) {
    id  = rs.getInt("id") ;
    age = rs.getInt("age") ;
    first = rs.getString("first") ;
    last = rs.getString("last") ;
    // display or process the retrieved data (i.e., variables id, age, first and last) here
}
```

Please note that when you create a Statement object it internally defines the behaviour of the resultset that will hold the returned results. Since, we are providing no parameter to *createStatement( )*, the resultset will have the default behaviour which is being described below.

The default behaviour of *ResultSet* objects is as follows :

(i) You can only retrieve data using appropriate getXXX( ) method.

(ii) You cannot go back in a *ResultSet* that means all are **forward only** resultsets.

There are ways to change the default behaviour of result set *i.e.*, the resultsets can be made *scrollable* and *read_only* also while creating[2] the *Statement* object, but we are not going into further details of this as it is beyond the scope of the book.

Step 6 : Clean up the Environment

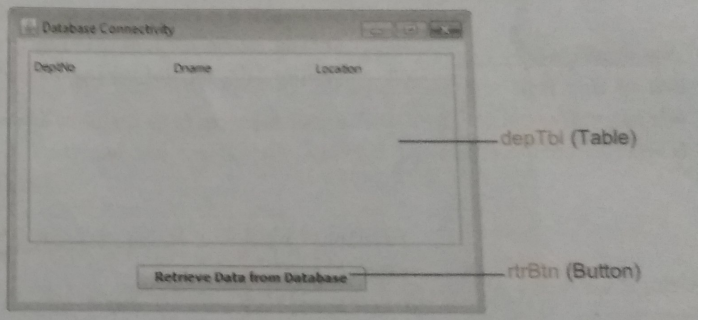
After all the processing, final step is to clean the environment. In this step you should explicitly close all database resources using close( ) method as shown below :

```
//STEP 6: Clean-up environment
rs.close() ;
stmt.close() ;
conn.close() ;
```

Armed with this knowledge, let us do this practically. Following example practically retrieves data from *dept* table of *test* MySQL database.

Practical Learning

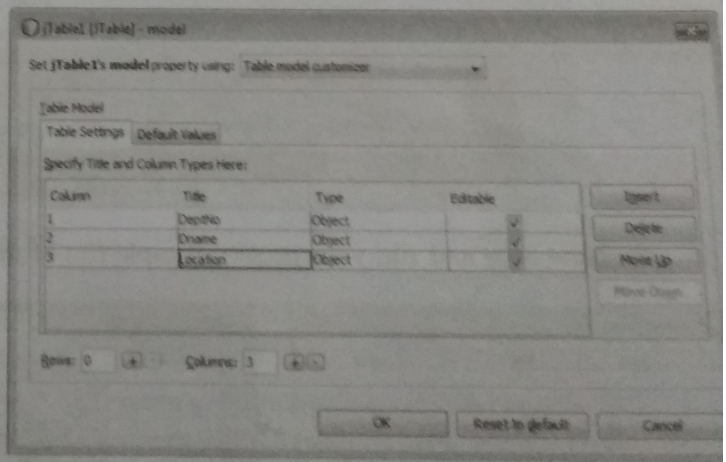
Example 8.1 Retrieving data from dept table from test MySQL database from within Java GUI application.



Solution.

1. Start NetBeans *Java Application* project, add a JFrame form to it.

2. Add a Table (namely **depTbl**) to the frame's design space. Edit its **model** property to set its number of **Rows** as 0 (Zero), **Columns** as 3 and the *Column Titles* as : *DeptNo, Dname, Location*. (see here)



3. Now add a button to the design space below the table as shown in the screenshot above. Name it as **rtrBtn**.

2. An example statement is being given here for your reference :

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

If you want to know further, you can do it yourself by referring to books on JDBC.

4. Now open the *Projects window* by pressing **Ctrl+1**. Expand the *Libraries* node of your project. Right click on the *Libraries* node and click on **Add JAR/Folder** option (as explained in Step 1(*b*) of section 8.4.1).
   Then select *mysql-connector-java bin file* after browsing your PC and click *Open*, so that it gets added to the *Library* node of your project.
5. In the Source editor of your project, type the following code line at the top line of it :

```
import java.sql.* ;          //step 1
```

6. Now type the following code in the Action event handler of *rtrBtn* button.

```
DefaultTableModel model = ( DefaultTableModel) depTbl.getModel() ;
try {
    Class.forName("java.sql.Driver") ;

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

    Statement stmt = con.createStatement() ;
    String query="SELECT * FROM dept ;" ;
    ResultSet rs = stmt.executeQuery(query) ;

    while(rs.next())  {
        String dno = rs.getString ("deptno") ;
        String dName = rs.getString ("dname") ;
        String lc = rs.getString ("loc") ;

        model.addRow (new Object[] {dno, dName, lc}) ;
    }

    rs.close() ;
    stmt.close() ;
    con.close() ;
}
catch(Exception e) {
    JOptionPane.showMessageDialog(null, "Error in connectivity") ;
}
```

Enclose your connectivity code in try{} otherwise you'll receive errors and won't be able to run your code

Step 2

Step 3

Step 4

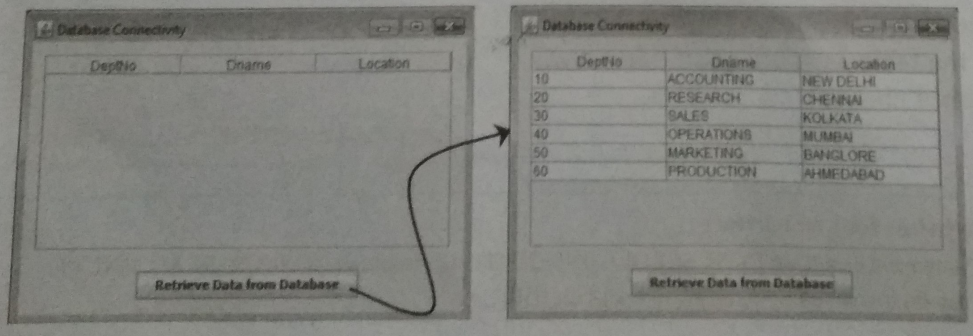
Step 5

This statement will first create an Object array from the data values fetched from the resultset and then add this into the table's model. This way all the rows fetched data would be added to depTbl one by one.

Step 6

A try{} must be followed by a catch() block that gets executed in case an exception(runtime error) is thrown

7. Now save and run your project. Sample run of this project is shown below :



NOTE

Please note, in place of user-id ("root" here) and password ("wxyz" here) given here, you have to write your own user-id and password that you use to access MySQL database on your machines.

8.4.2 ResultSet Methods

A *ResultSet* object maintains a cursor, which points to its current row of data. When a *ResultSet* object is first created, the cursor is positioned before the first row. To move the cursor, you can use any of the following methods. (Say our ResultSet object's name is *rs*.)

- **next( )** — moves the cursor forward one row. Returns *true* if the cursor is currently positioned on a row and *false* if the cursor is positioned after the last row.
- **first( )** — moves the cursor to the first row in the ResultSet object. Returns *true* if the cursor is currently 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 currently positioned on the last row and *false* if the *ResultSet* object does not contain any rows.
- **relative(int rows)** — moves the cursor relative to its current position. *e.g.*, if the cursor is currently at record 3 then rs.relative(2) will place the cursor at 5th record.
- **absolute(int rno)** — positions the cursor on the rno-th row of the ResultSet object, *e.g.*, rs.absolute(2) will place the cursor at 2nd record irrespective of its current position.
- **getRow( )** — Retrieves the current row number the cursor is pointing at. That is, if cursor is at first row the *getRow*( ) will return 1.

Once you have a scrollable *ResultSet* object, you can use it to move the cursor around in the result set. Since the cursor is initially positioned just above the first row of a *ResultSet* object, the first call to the method next( ) moves the cursor to the first row and makes it the current row. Successive invocations of the method next( ) move the cursor down one row at a time from top to bottom. You have already learnt how it works in the code examples earlier in the chapter.

8.4.3 Retrieving Data based on User-specified Criteria

In the example that we discussed just before, you learnt how to retrieve data from a database's table from within Java application. The query that retrieved data from database was a simple one. However, you can also execute complex queries even if they are based on user -specified criteria. Let's see how. For instance, you want to obtain those records from a table (say *employee*) where *salary* is more than a value specified by the user in a *textfield* namely *salTF*.

For this , you may frame your SQL query as follows :

eg: (`String query = "SELECT * from employee WHERE salary >"+ salTF.getText() +" ;" ;`)

This will concatenate user-specified criteria with the SQl query.

This will concatenate semicolon with the query. Without semicolon, SQL query is not complete.

You may even obtain the field-name and/or comparison operator along with the criteria from the user. All you need to do is to concatenate the obtained values in the SQL query at appropriate place along with proper blank spaces in between. (Workshop 8.1 does the same). *Don't forget to concatenate semicolon at the end of the query. Also, it is suggested that you display the query string before executing it so that you get to see whether punctuators like quotes and semicolon have been properly concatenated or not.*

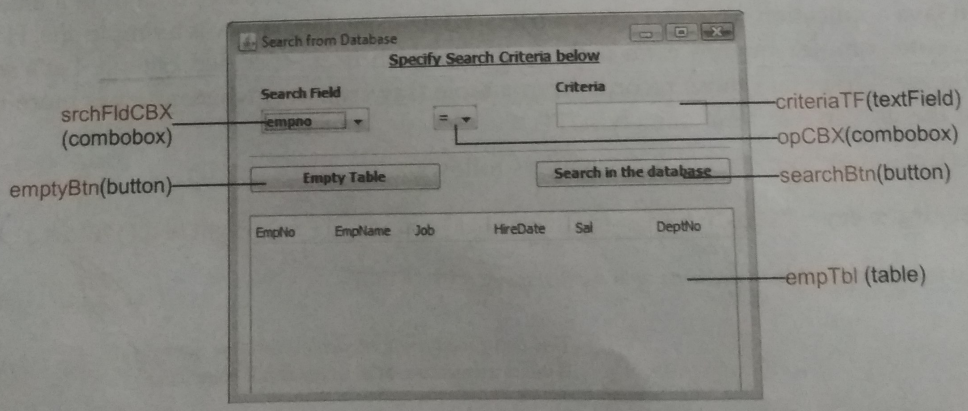
Let us now move on to workshop 8.1 that retrieves data from *test* database's *empl* table based on a user specified criteria.

The *empl* table stores following data :

empno	ename	job	mgr	hiredate	sal	comm	deptno
8369	Smith	Clerk	8902	18 Dec, 1990	800.00	<NULL>	20
8499	Anya	Salesman	8698	20 Feb, 1991	1600.00	300.00	30
8521	Seth	Salesman	8698	22 Feb, 1991	1250.00	500.00	30
8566	Mahadevan	Manager	8839	2 Apr, 1991	2985.00	<NULL>	20
8654	Momin	Salesman	8698	28 Sep, 1991	1250.00	1400.00	30
8698	Bina	Manager	8839	1 May, 1991	2850.00	<NULL>	30
8839	Amir	President	<NULL>	18 Nov, 1991	5000.00	<NULL>	10
8844	Kuldeep	Salesman	8698	8 Sep, 1991	1500.00	0.00	30
8882	Shiavnsh	Manager	8839	9 Jun, 1991	2450.00	<NULL>	10
8886	Anoop	Clerk	8888	12 Jan, 1993	1100.00	<NULL>	20
8888	Scott	Analyst	8566	9 Dec, 1992	3000.00	<NULL>	20
8900	Jatin	Clerk	8698	3 Dec, 1991	950.00	<NULL>	30
8902	Fakir	Analyst	8566	3 Dec, 1991	3000.00	<NULL>	20
8934	Mita	Clerk	8882	23 Jan, 1992	1300.00	<NULL>	10

Workshop 8.1 Searching in the Database

Create a Java GUI application that obtains the search criteria from the user and retrieves data from the database based on that.



1. Create a new Java Application project in NetBeans IDE. Add a JFrame form to it.

2. Now add some labels as per the given screenshot and specify their text properties as per the screenshot given above.

3. Add two combo boxes namely srchFldCBX and opCBX respectively and set their *model* properties as shown below :

srchFldCBX's model	opCBX's model
empno	=
ename	<
job	>
sal	!=
deptno	

4. Add a text field namely criteriaTF and then add two buttons with *text* "Empty Table" (emptyBtn) and "Search in the database" (searchBtn).

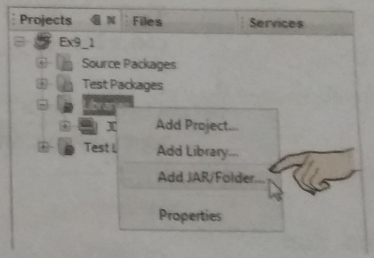
5. Add a table with six columns with headings as shown in the screenshot. Set its *Rows* to 0 and name this table as empTbl.

Import Required Classes

6. At the top of all the source code in the code-editor, type the following import commands :

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

7. Open **Projects** window by pressing Ctrl+1. Expand the project node and then **Libraries** node beneath it. Right click the **Libraries** node and click Add JAR/Folder... command to select the *mysql-java-connector's bin file*. (See figure on the right side).



Add functionality to *Empty Table* button

8. Write following code in the *Action event handler* of the button namely *EmptyBtn* to remove all the rows form the table :

```
DefaultTableModel model = (DefaultTableModel) empTbl.getModel() ;
int rows = model.getRowCount( ) ;
    // Now remove all the rows from the current table model
if (rows > 0) {
    for (int i = 0 ; i<rows ; i++) {
        model.removeRow(0) ;
    }
}
```

Remove 0th row every time because after removal, every next row becomes 0th row

Add functionality to *Search in the database* button

9. Write the following code in the *Action event handler* of the button namely searchBtn :

```
emptyBtn.doClick() ;
DefaultTableModel model = ( DefaultTableModel) empTbl.getModel() ;
try {
    Class.forName("java.sql.Driver") ;
    Connection con = DriverManager.getConnection
        ("jdbc:mysql://localhost/test","root","wxyz") ;
    Statement stmt = con.createStatement() ;
    String sfld = (String) srchFldCBX.getSelectedItem() ;
    String op = (String) opCBX.getSelectedItem() ;
    String crit = criteriaTF.getText() ;
    String query= "SELECT empno, ename, job, hiredate, sal, deptno
                   FROM empl WHERE" + sfld + " " + op + " " + crit + " ;" ;

    ResultSet rs = stmt.executeQuery(query) ;
    while(rs.next())  {
        model.addRow (new Object[ ] {
            rs.getInt (1), rs.getString (2), rs.getString (3),
            rs.getDate(4),rs.getFloat(5), rs.getInt(6)
        } ) ;
    }
    rs.close() ;
    stmt.close() ;
    con.close() ;
}
catch(Exception e) {
    JOptionPane.showMessageDialog(null, "Error in connectivity") ;
}
```

This would invoke Empty Table button to empty the contents of the table

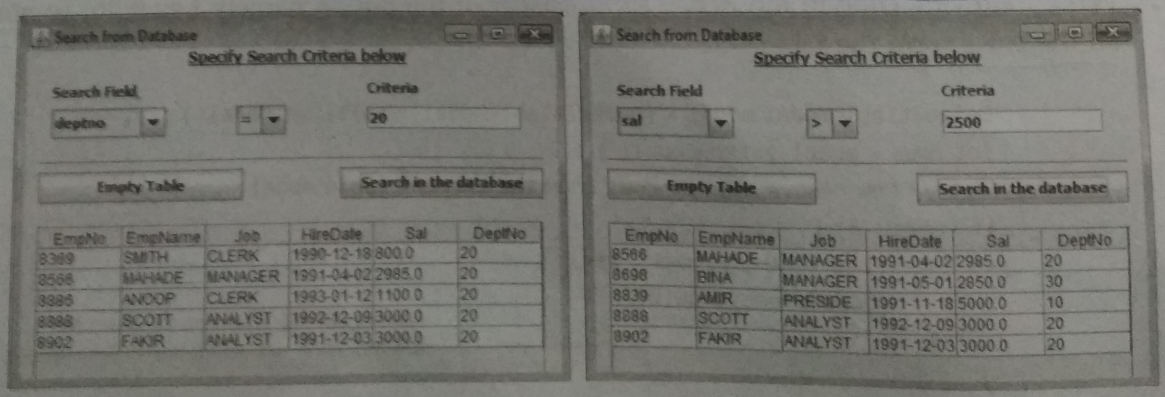
Here take care to concatenate quotes surrounding criteria, if needed i.e., with string values and data values

SQl query framed using values obtained from GUI fields.

In this statement, the data is directly fetched from the recordset using their getXXX() methods and then the retrieved row is cast to Object array before adding into table's model.

Save and Run

10. Save and run your project. The sample run of this application is shown below. Please note that while specifying string values in criteria text field, make sure to enclose them in quotes.



8.4.4 Performing IUD (Insert/Update/Delete) Queries

As we have earlier mentioned that if your query is not a SELECT query but on INSERT / DELETE / UPDATE query then in place of **<statement-object>.executeQuery( )** method, you need to invoke **<statement-object>.executeUpdate( )** method.

For instance, if you are obtaining values from three text fields (*dnoTF*, *dnameTF*, *locTF*) and intend to insert a row with these values as fields in the database table *dept*, then you may write code as :

```
:   //driver registered and connection (conn) opened
String query = "INSERT INTO dept VALUES ( "
                    + dnoTF.getText( )
                    + " ' " + dname.getText( ) + " ' "
                    + " ' " + locTF.getText( ) + " ' "
                    + " ) ; " ;
Statement stmt = conn.createStatement( ) ;

ResultSet rs = stmt.executeUpdate(query) ;
```

See quotes are being concatenated carefully

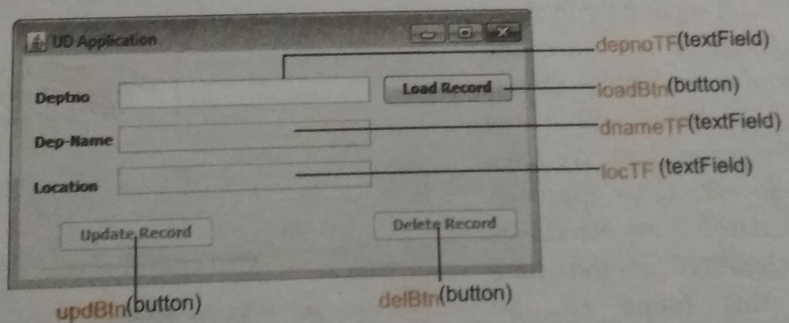
See we have concatenated quotes for string values

As you can make out that while framing queries, you need to make sure that spaces, quotation marks and semicolon must be concatenated at appropriate places. If you want to be sure that correct sql query is framed, you can display the *query* string before executing it, ONLY during the developing phase. Once your application starts working correctly, you can do-away with query-display-statement.

Following workshop 8.2 is based on an application that runs two non-SELECT sql queries. So, let's move on to that.

Workshop 8.2 Updating / Deleting in the Database

Create a Java GUI application that performs Update and Delete operations on a database. Update and Delete buttons should get activated only if the record based on provided search value is found in the database.



1. Create a new Java Application project in NetBeans IDE. Add a JFrame form to it.

2. Now add some labels as per the given screenshot and specify their text properties as per the screenshot given above.

3. Add three text fields namely **depnoTF**, **dnameTF** and **locTF**. Apart from setting other properties, set **enabled** property of *dnameTF* and *locTF* as *false* initially.

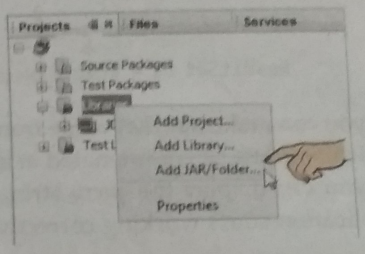
4. Add three buttons with *text* "Load Data" (**loadBtn**), "Update Record" (**updBtn**) and "Delete Record" (**delBtn**).

Import Required Classes

5. At the top of all the source code in the code-editor, type the following import commands :

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

6. Open **Projects** window by pressing Ctrl+1. Expand the project node and then **Libraries** node beneath it. Right click the **Libraries** node and click **Add JAR/Folder**... command to select the *mysql-java-connector's bin file*. (See figure on right side).

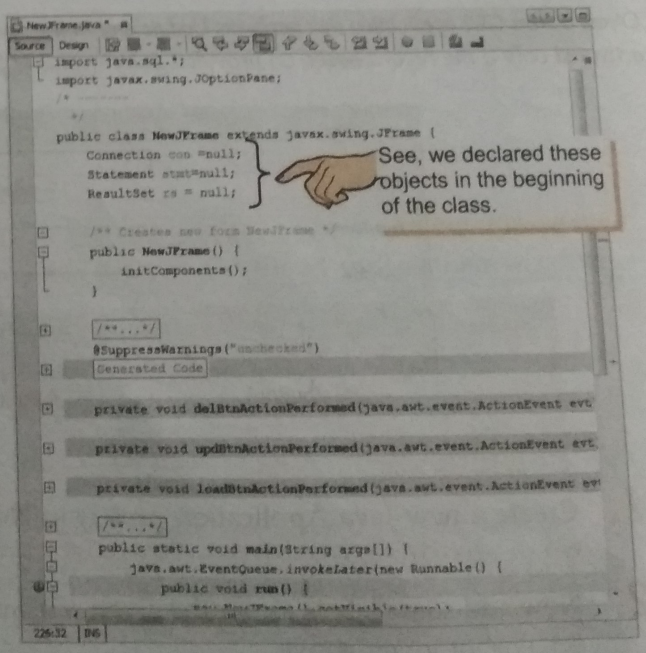


Declare Data Connectivity object

7. When we need to perform database connectivity operations in multiple methods *e.g.* here we need to perform three different database-connectivity operations in the even-handler methods of three buttons *i.e., Load-data, Update-record* and *Delete-record*.

For such a case, we can ***either*** declare and create Data-base connectivity objects (Connection, Statement and ResultSet objects) separately in these different methods ***or*** we can create these objects once and those objects can be used globally within the methods of the same class. We have decided to follow the latter approach so, we first declared these objects as the first declaration in the class representing this frame *i.e., we declared these objects as* :

```
Connection con = null ;
Statement stmt = null ;
ResultSet rs = null ;
```



That is, now our source editor windows looked somewhat like :

Add functionality to *Load Data* button

8. Write the following code in the *Action* event handler of the button namely *loadBtn* :

```
try {
    Class.forName("java.sql.Driver") ;
    con = DriverManager.getConnection("jdbc:mysql://localhost/test","root","wxyz") ;
    stmt = con.createStatement() ;
    String query= "SELECT * FROM dept WHERE deptno =" + depnoTF.getText()+";" ;
    rs = stmt.executeQuery(query) ;
    if ( rs.next()) {
        String dno = rs.getString ("deptno") ;
        String dname = rs.getString ("dname") ;
        String lc = rs.getString ("loc") ;
        dnameTF.setText(dname) ;
        locTF.setText(lc) ;
        updBtn.setEnabled(true) ;
        delBtn.setEnabled(true) ;
        dnameTF.setEnabled(true) ;
        locTF.setEnabled(true) ;
    }
    else {
        JOptionPane.showMessageDialog(null,"No such record found.") ;
    }
}
catch(Exception e) {
    JOptionPane.showMessageDialog(null, "Error in connectivity") ;
}
```

Add Functionality to *Update Record* button

9. Write the following code in the *Action* event handler of the button namely *updBtn*. Notice that this code is neither *registering the driver* nor *opening the connection* because it will be using the same connection that has already been opened in the *Action* event of *Load data* button.

```
int ans = JOptionPane.showConfirmDialog(null, "Surely wanna update the record?") ;
if (ans == JOptionPane.YES_OPTION) {
  try {
      stmt = con.createStatement() ;
      String query = "UPDATE dept SET dname = '"+dnameTF.getText() +"',
             loc = '"+locTF.getText()+"' WHERE deptno = " +depnoTF.getText()+" ;" ;
      stmt.executeUpdate(query) ;
      JOptionPane.showMessageDialog(null, "Record successfully updated.") ;
  }
  catch(Exception e) {
      JOptionPane.showMessageDialog(null,"Error in table updation!") ;
  }
  depnoTF.setText("") ;
  dnameTF.setText("") ;
  locTF.setText("") ;
  updBtn.setEnabled(false) ;
  delBtn.setEnabled(false) ;
} // end of if
```

If Yes button is pressed then

con is already opened in action event of load data

See, executeUpdate() is being used here

These statements would clear the text-field and disable the update and delete buttons.

Add functionality to *Delete Record* button

10. Write the following code in the *Action* event handler of the button namely *delBtn*. Like *updBtn's* code, this code is neither *registering the driver* nor *opening the connection* because it will also be using the same connection that has already been opened in the *Action* event of *Load data* button.

```
int res = JOptionPane.showConfirmDialog(null, "Wanna delete the record for sure?") ;
if (res == JOptionPane.YES_OPTION) {
    try {
                                        con is already opened in action event of load data
        stmt = con.createStatement() ;
        String query = "DELETE FROM dept WHERE deptno = "+depnoTF.getText()+ ";" ;
        stmt.executeUpdate(query) ;
        JOptionPane.showMessageDialog(null,"Record deleted!") ;
    }
    catch(Exception e) {
        JOptionPane.showMessageDialog(null,"Error in deletion!") ;
    }
} // end of if
depnoTF.setText(" ") ;
dnameTF.setText(" ") ;
locTF.setText(" ") ;
updBtn.setEnabled(false) ;
delBtn.setEnabled(false) ;
```

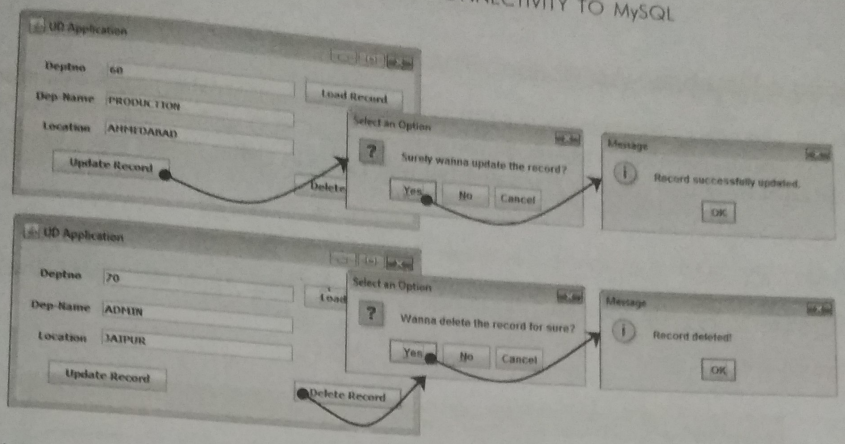
Save and Run

11. Save and run your project. The sample run of this application is shown below.

The dept table initially contains :

```
mysql> SELECT * FROM dept ;
+--------+------------+-------------+
| deptno | dname      | loc         |
+--------+------------+-------------+
|    10  | ACCOUNTING | NEW DELHI   |
|    20  | RESEARCH   | CHENNAI     |
|    30  | SALES      | KOLKATA     |
|    40  | OPERATIONS | MUMBAI      |
|    50  | MARKETING  | BANGLORE    |
|    60  | PRODUCTION | CHANDIGARH  |
|    70  | ADMIN      | JAIPUR      |
+--------+------------+-------------+
7 rows in set (0.00 sec)
```

We modified record with deptno as 60 by changing its location to AHMEDABAD and deleted record with deptno as 70.



The dept table after these operations contained following data :

```
mysql> SELECT * FROM dept ;
+--------+------------+-------------+
| deptno | dname      |    loc      |
+--------+------------+-------------+
|     10 | ACCOUNTING |   NEW DELHI |
|     20 | RESEARCH   |   CHENNAI   |
|     30 | SALES      |   KOLKATA   |
|     40 | OPERATIONS |   MUMBAI    |
|     50 | MARKETING  |   BANGLORE  |
|     60 | PRODUCTION |   AHMEDABAD |
+--------+------------+-------------+
6 rows in set (0.00 sec)
```

Let Us Revise

- *To connect to a database from within a programming application, you need a framework that facilitates communication between two different generes of software (programming application and DBMS).*
- *Some most popular database connectivity frameworks include ODBC (Open DataBase Connectivity) and JDBC (Java DataBase Connectivity).*
- *ODBC is a framework for accessing Microsoft supported databases whereas JDBC is a framework for accessing databases from within Java-applications.*
- *Four main JDBC classes that are used for database connectivity are : DriverManager class, Connection class, Statement class and ResultSet class.*
- *Steps to create database connectivity application are :*
  - *(i) import the packages required for database programming by using (a) import command and (b) add database-connector Jar file to your project.*
  - *(ii) Register the JDBC driver using* ***Class.forName( )*** *method.*
  - *(iii) Open the connection.* *(iv) Execute the query.*
  - *(v) Process the resultset.* *(vi) Clean up the environment.*
- *To connect to MySQL from Java, two commonly used drivers are : java.sql.Driver and com.mysql.jdbc.driver.*
- *For executing SELECT queries, executeQuery( ) method is used whereas for executing INSERT/DELETE/UPDATE queries, executeUpdate( ) method is used.*