Java SQL

Communicating with Databases

web explorations

Introduction

SQL - Standard Query Language

SQL is a special-purpose programming language used to communicate with relational databases such as mySQL, Microsoft Access, Oracle, Microsoft's SQL Server, MariaDB, and DB2.

It is utilized by virtually all popular computer languages (including COBOL) making it a critical skill to have experience with.

DBF - DataBase File
A database file resides on a database server and contains various tables. Each table is made up of records (rows) and fields (columns).

A Little History

SQL was initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s. This version, initially called SEQUEL (Structured English Query Language), was designed to manipulate and retrieve data stored in IBM's original database management system.
— Wikipedia http://en.wikipedia.org/wiki/SQL

How should I pronounce it? SQL or ESS QUE EL?

Pat RJK was thinking about the on-going discussion of how to pronounce SQL. Here is what he wrote in his blog:
Then a thought occurred to me: SQL was created in the 70's, the creators are probably techies, I can probably just email them and ask them how it's pronounced! Ray Boyce had passed away at a young age, but Don Chamberlin was alive and now teaching at a university. I felt a little silly, but I decided to fire off a short email to him:
Hello Don,

I'm sorry to waste your time with such a silly question, but I've often heard SQL pronounced S-Q-L or as Sequel. I've also seen the official pronunciation listed both ways. According to wikipedia, you and Raymond Boyce created the language and it was shortened to SQL after some legal dispute. So my question is, is there an official pronunciation to SQL? Thank you for your time.
– Pat
To my delight, he replied back:
Hi Pat,
Since the language was originally named SEQUEL, many people continued to pronounce the name that way after it was shortened to SQL. Both pronunciations are widely used and recognized. As to which is more "official", I guess the authority would be the ISO Standard, which is spelled (and presumably pronounced) S-Q-L.
Thanks for your interest,
Don Chamberlin

— http://patorjk.com/blog/2012/01/26/pronouncing-sql-s-q-l-or-sequel/

***

The important thing is when you are on a job interview if someone asks if you know SQL, no matter how they pronounce it (S-Q-L or sequel) you should automatically know that these are the same thing.

SQL Statements

There are over 800 SQL keywords but you only need to be familiar with a few of these.

This tutorial will demonstrate the most common SQL statements using Java.

You will learn how to:

- CREATE TABLE - Create an new, empty table.
- INSERT a new record of data into a table.
- SELECT a set of data from a table.
- UPDATE any record in a table.
- DELETE a record from a table.
- DROP (delete) a table.

These make up the most common SQL functions summarized with the acronym, CRUD:

- Create - INSERT
- Read - SELECT
- Update - UPDATE
- Delete - DELETE

You'll also become familiar with some of the Java methods from the DriveManager and Statement classes that allow you to do the heavy lifting:

DriveManager.getConnection( ) - Connect your Java program to a database. We will be using mySQL, but with a small edit and the correct software driver you can connect to any relational database.

Statement.executeUpdate( ) - Run SQL commands that manipulate the database tables but don't return any data. This would include CREATE, DROP, DELETE, UPDATE, and INSERT.

Statement.executeQuery( ) - Run SQL commands that return a result set. This would include SELECT statements.

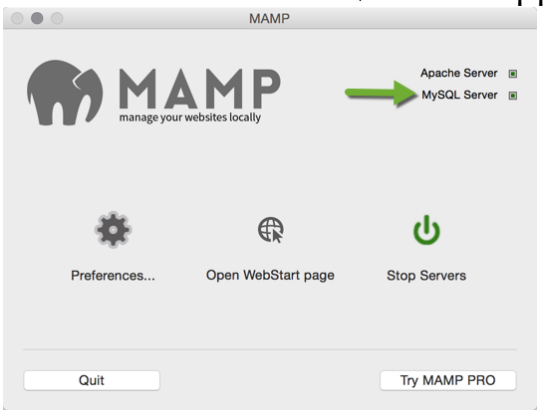
Getting the DBF Setup

Install MAMP, WAMP, or XAMPP on your computer.

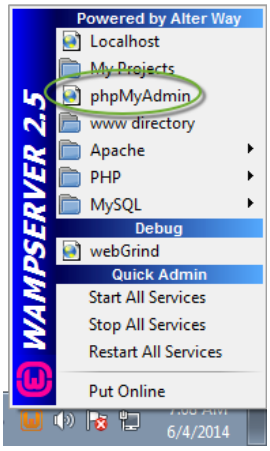
Each of these will give you a web server stack: Apache web server, mySQL database server, and PHP.
Once the package is installed and running you can use Java (instead of the built-in PHP) to work with the mySQL database server.

Depending on which package you use there will be a different control panel. Keep in mind that software is constantly being updated and your control panel may appear slightly different than these screen shots.

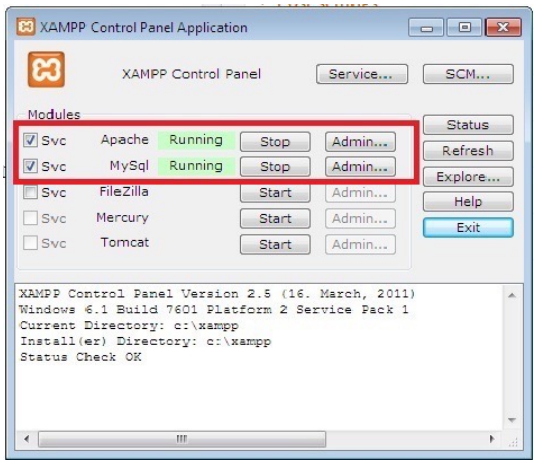
MAMP Control Panel (Run the application to see the window)



WAMP Control Panel (click on the WAMP icon in the toolbox)



XAMPP Control Panel (Click on the icon on the desktop)

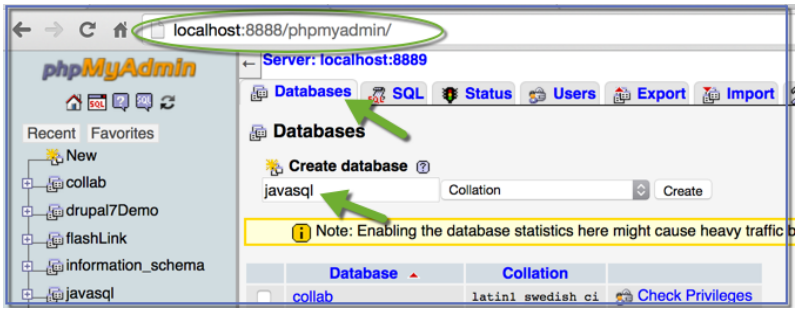


Create a new database on your mySQL server.

phpMyAdmin is a GUI (Graphical User Interface) that allows you to work with the data inside the mySQL database server.

For WAMP and XAMPP click on the appropriate link in the control panel to access phpMyAdmin.

For MAMP, open up a web page and type in this URL: http://localhost:8888/phpmyadmin



Click on the database tab and type in the name for the new database.

javasql is a good name for this project.

All you need now is a table. We will use Java to add in a new table and fill it with data.

Getting Eclipse Setup

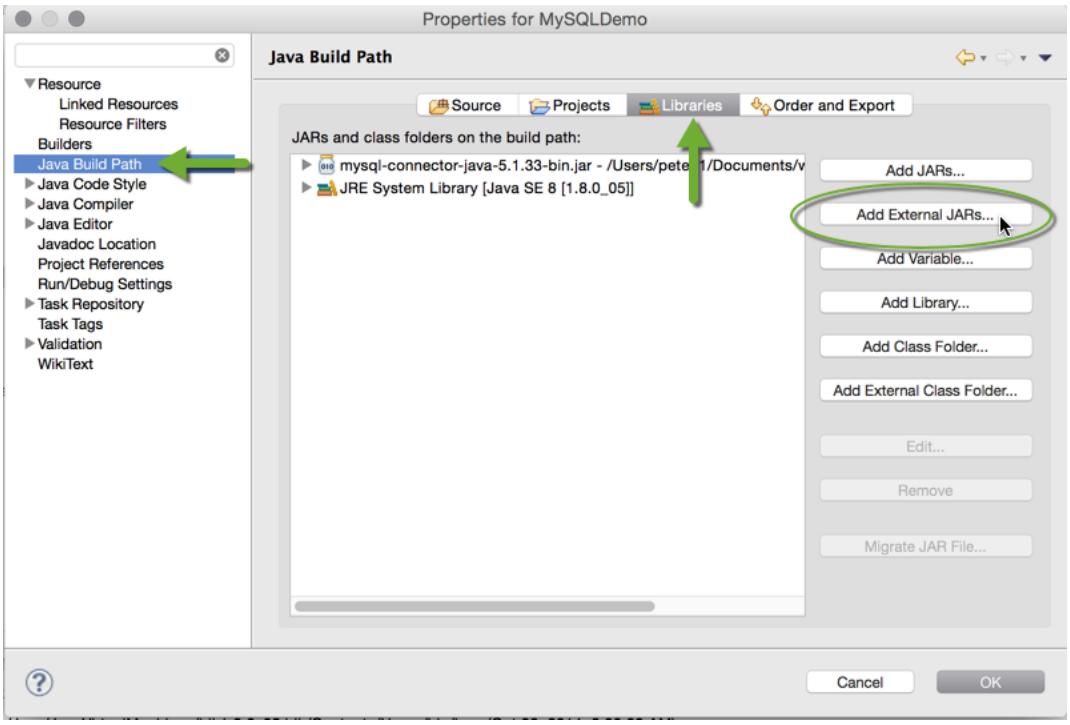
Install the mySQL Driver in Eclipse.

Java needs a driver, or software bridge or translator, connecting it to the database server.
mySQL offers the software you need, call the Connector J here: http://dev.mysql.com/downloads/connector/j/

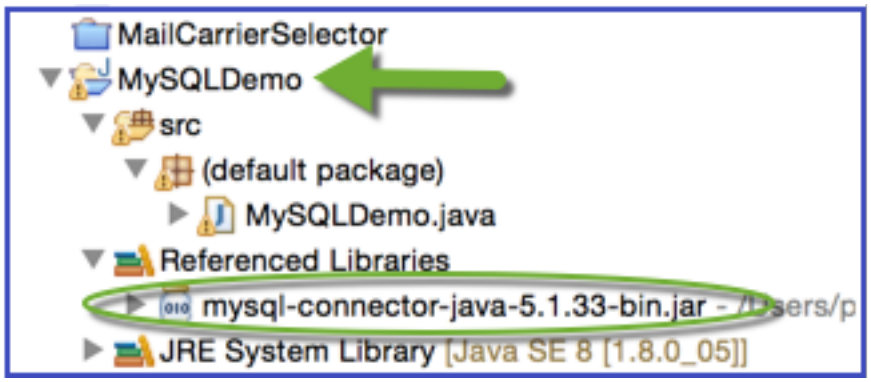
Download the Driver
Download it and uncompress the file on your desktop or download folder.

Open up Eclipse and create a new project. You might want to name it **MySQLDemo.**
Copy the file named mysql-connector-java-5.x.xx-bin.jar into the root of the project file. (The version number designated by the x.xx will probably be different when you download the file).

Tell Eclipse to include this driver as part of the file.
Select Project from the menu bar.
Select "Java Build Path" from the left menu bar.
Click on the "Libraries" tab.
Click on the "Add External JARs" button on the right.
Navigate to the actual file located in the Eclipse project folder and select it.



When complete your Eclipse Package Explorer should look something like this:



The mysql-connector-java-5.x.xx-bin.jar file should be in the “Reference Library” folder.

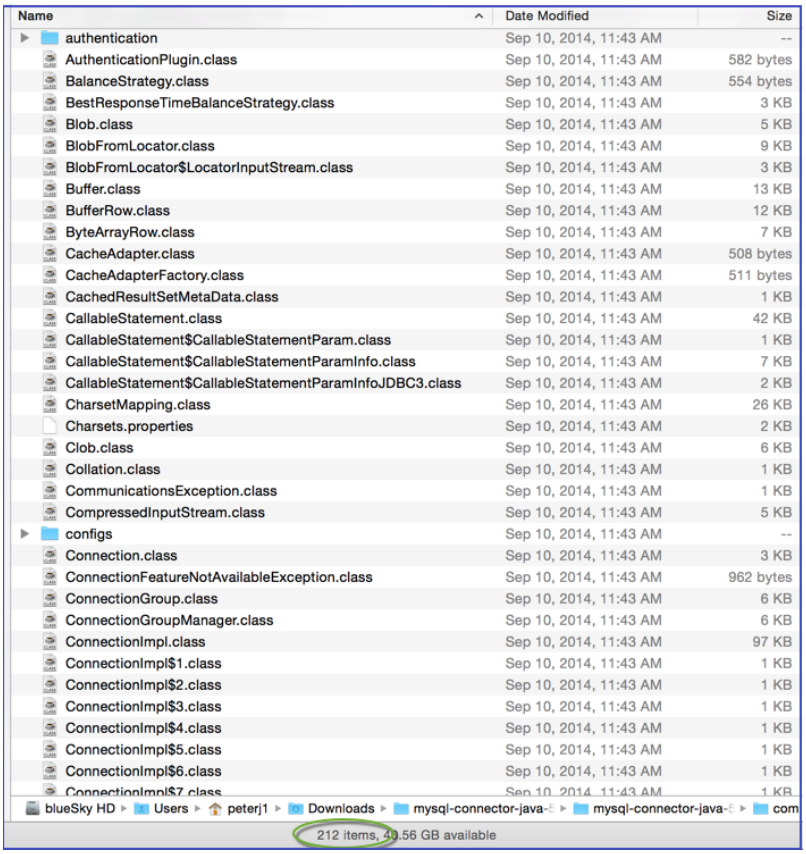
What is a .jar file?

A jar file is very similar to a .zip or .gz file. It is a series of classes and support files compressed into a single file.

The sweet thing about a .jar is that it doesn’t have to be unjarred (uncompressed). Java can access the compressed files and use them

automatically, keeping everything neat and tidy.

For example, if you uncompressed mysql-connector-java-5.x.xx-bin.jar you would find three main folders and over 212 .class files.



Write the Utility Methods

Before we write any test code we need to include some utility methods.

Once these methods are set up and running they don't need a lot of maintenance so they are usually put at the very bottom of the class.

We have three methods that will be doing the heavy lifting:

1. getConnection( )
2. executeUpdate( )
3. releaseResource( )

getConnection( ) will connect the program to the database on the mySQL server.

```
/* ********************************
 * HEAVY LIFTING - Used by all the CRUD methods.
 **********************************/
 /**
 * getConnection( ) - Get a new database connection
 *
 * @return Connection
 * @throws SQLException
 */
 public Connection getConnection() throws SQLException
 {
    Connection conn = null;
    Properties connectionProps = new Properties();
    connectionProps.put("user", this.userName);
    connectionProps.put("password", this.password);

    conn = DriverManager.getConnection("jdbc:mysql://"
            + this.serverName + ":" + this.portNumber + "/" + DBF_NAME,
            connectionProps);
    return conn;
 } // end of getConnection( )
```

This is standard code used to connect Java with the database server. What makes this code so important is that the getConnection( ) parameters can be easily changed to accommodate different databases such as Oracle, Access, etc.

Each time we need to access the database we open up a connection, do the work, and immediately close the database. Consequently this method gets called a lot.

executeUpdate( ) will send SQL statements to the database server.

```
/**
 * executeUpdate( ) - Used to run a SQL command which does NOT return a resultSet:
 * CREATE/INSERT/UPDATE/DELETE/DROP
 *
 * @throws SQLException If something goes wrong
 * @return boolean if command was successful or not
 */
public boolean executeUpdate(Connection conn, String command) throws SQLException
{
    Statement stmt = null;
    try
    {
        stmt = conn.createStatement();
        stmt.executeUpdate(command); // This will throw a SQLException if it fails
        return true;
    }
    finally
    {
        // This will run whether we throw an exception or not
        if (stmt != null) { stmt.close(); }
    }
} // end of executeUpdate( )
```

Once we have the connection we need to "talk" to the database. There are two methods that communicate via SQL statements.
executeUpdate( ) is used when the SQL statements are not expecting any data back. This would include the CREATE, INSERT, UPDATE, DELETE, and DROP SQL statements. They change the tables but do not return a result set (rs).

Later you will be adding an **executeQuery( )** method used with the SELECT statements which will return a result set (rs).

Here is the code for releaseResource( ):

```
    /**
     * releaseResource( ) - Free up the system resources that were opened.
     *                      If not used,  a null will be passed in for that parameter.
     * @param rs - Resultset
     * @param ps - Statement
     * @param conn - Connection
     */
    public void releaseResource(ResultSet rs, Statement ps, Connection conn )
    {
       if (rs != null)
       {
          try { rs.close(); }
          catch (SQLException e) { /* ignored */}
       }
       if (ps != null)
       {
          try { ps.close(); }
          catch (SQLException e) { /* ignored */}
       }
       if (conn != null)
       {
          try { conn.close();}
          catch (SQLException e) { /* ignored */}
       }
    } // end of releaseResource( )

} // end of class MySQLDemo
```

Database resources are expensive so we need to release the connections each time we are finished using them. On a live site this will allow other users to access the data without waiting.

Testing the Connection with main( )

Write some test code.

We need two methods to do the heavy lifting and a main( ) method.
You can put these anywhere inside the class but you might find this order to be the easiest for maintaining your code.

1. CONSTANTS
2. DBF Credentials
3. main( )
4. CRUD methods will go next (normally require the most editing and are commonly organized in alphabetical order.)
5. getConnection( ), executeUpdate( ), and releaseResource( ) methods at the bottom of the class. (Do the heavy lifting but not edited too much.)

CONSTANTS and main( )

At the top of the class set up two constants for the database name and table name.
Create the global variables necessary to access the database
In the main( ) create a new MySQLDemo object.

```
public class MySQLDemo
{
    // Set up two constants to be used in this test environment.
    private final static String DBF_NAME = "javasql";
    private final static String TABLE_NAME = "customer";

    // The mySQL username and password (may be empty)
    private final String userName = "root";
    private final String password = "root";
    // Name of the computer running mySQL
    private final String serverName = "localhost";
    // Port of the MySQL server (default is 3306 or 8889 on MAMP)
    private final int portNumber = 8889;

    public static void main(String[] args)
    {
        MySQLDemo app = new MySQLDemo();
        app.createTable(TABLE_NAME);
    } // end of main( )
```

Constants

DBF Credentials

main( )

Could we make the login credentials constants? Absolutely.
Note that we haven't written the createTable( ) method yet. That comes next….

createTable( )

Here is the code for the basic createTable( ) method.

```
public void createTable(String tableName)
{
    Connection conn = null;
    String sql = "";

    // Connect to MySQL
    try
    {
       conn = this.getConnection();
    }
    catch (SQLException e)
    {
        System.out.println("ERROR: Could not connect to the database");
        e.printStackTrace();
        return;
    }

    // Create a table
    try
    {
       sql = "CREATE TABLE " + tableName + " ( " +
         "id INTEGER NOT NULL AUTO_INCREMENT, " +
         "name varchar(40) NOT NULL, " +
         "street varchar(40) NOT NULL, " +
         "city varchar(20) NOT NULL, " +
         "state char(2) NOT NULL, " +
         "postalcode char(5), " +
         "PRIMARY KEY (id))";
       this.executeUpdate(conn, sql);
       System.out.println("Created table named:" + tableName);
    }
    catch (SQLException e)
    {
        System.out.println("ERROR: Could not create the table named: " + tableName);
        e.printStackTrace();
        return;
    }
    // release the resources
    finally { releaseResource(null, null, conn); }
} // end of createTable( )
```

Here is the process (and you'll see this repeated for all of the CRUD commands.)

1. Open a connection to the database using getConnection( )
2. Set up an SQL statement.
3. Send the SQL statement to the database server using executeUpdate( )

passing in the connection and SQL string as parameters.

4. Release the resources using releaseResource( ). If a resource was not used simply use null as the parameter.

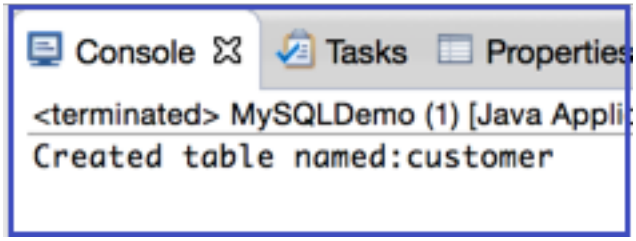
Some mySQL Database Tricks

The SQL statement CREATE TABLE includes the keyword AUTO_INCREMENT for the id field. And, at the end of the SQL statement you will notice where the id was set as the primary key. This means the table will be indexed or sorted on the id field.

By using AUTO_INCREMENT we are having the database server keep track of the next record item so we know that each id will be unique and we don't have to write a lot of code to deal with that.

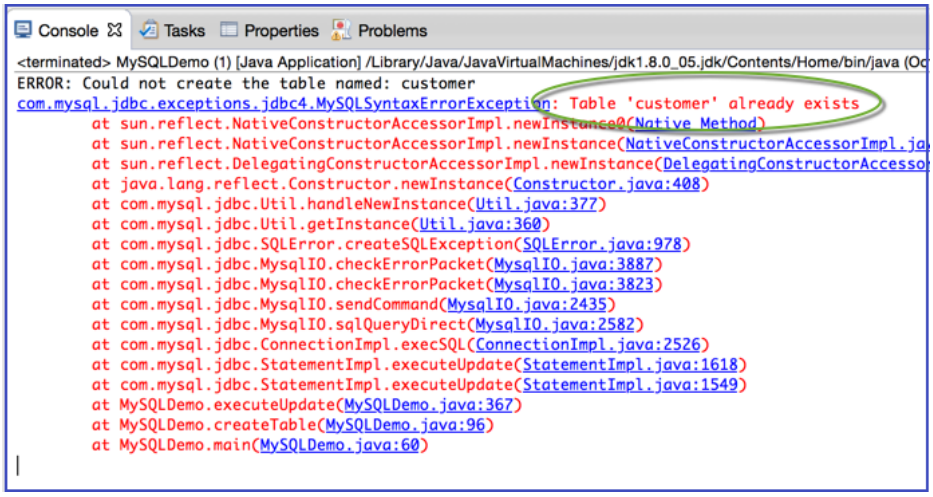
Also notice that all of the fields are Strings (varchar and char) except for the id which is an int so it can be auto incremented.

The first time you run the program you should get a result like this:



And, if you look in your phpMyAdmin (Don't forget to refresh the screen!) you should see a new table inside the database.

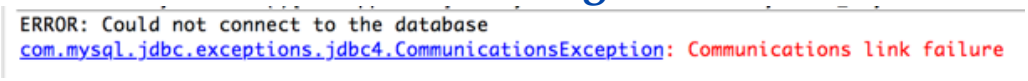
The next time you run the program there is already a table so you will get an nasty error message similar to this:



Don't let all the red information intimidate you. Look for the clues!
Ahh, the table already exists! Of course, I knew that!! (as you hit the heel of your hand against your forehead….)

In a little bit we will add code that will keep this error from happening.
But first, here are some common errors (and their solution) that you might run into.

3 Common DBF Error Messages



ERROR: Could not connect to the databasecom.mysql.jdbc.exceptions.jdbc4.CommunicationsException: Communications link failure

Make sure your MAMP, WAMP, or XAMPP is running ;-)

ERROR: java.sql.SQLException: No suitable driver found for jdbc:mysql://localhost:3306/

The JDBC Driver is not on the path. Eclipse is not aware of it.

To add it to your class path:

1. Right click on your project
2. Go to Build Path -> Add External Archives...
3. Select the file mysql-connector-java-5.1.24-bin.jar

NOTE: The version number in the .jar filename may be different if you have a newer version.

ERROR: java.sql.SQLException: Access denied for user 'userName'@'localhost' (using password: YES)

The userID and password are wrong.

MAMP: The user name and password are both "root" or "root".

WAMP: The user name is “root” and the password is empty “”

XAMPP: The user name is “user” and the password is what you typed in when you installed XAMPP.

Got a different error?

Do a web search using the text of the error message.

Only include the common error message and not any text that is unique to your machine such as filenames or database names.

createTable( ) with Error Checking

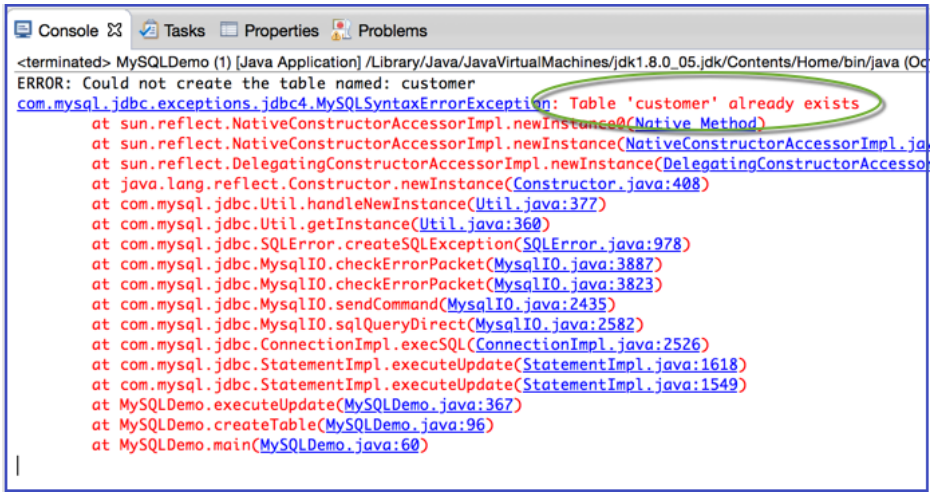
Here is the original code for the basic createTable( ) method that we added to our class a few pages back.

```
public void createTable(String tableName)
{
    Connection conn = null;
    String sql = "";

    // Connect to MySQL
    try
    {
       conn = this.getConnection();
    }
    catch (SQLException e)
    {
        System.out.println("ERROR: Could not connect to the database");
        e.printStackTrace();
        return;
    }

    // Create a table
    try
    {
       sql = "CREATE TABLE " + tableName + " ( " +
         "id INTEGER NOT NULL AUTO_INCREMENT, " +
         "name varchar(40) NOT NULL, " +
         "street varchar(40) NOT NULL, " +
         "city varchar(20) NOT NULL, " +
         "state char(2) NOT NULL, " +
         "postalcode char(5), " +
         "PRIMARY KEY (id))";
       this.executeUpdate(conn, sql);
       System.out.println("Created table named:" + tableName);
    }
    catch (SQLException e)
    {
        System.out.println("ERROR: Could not create the table named: " + tableName);
        e.printStackTrace();
        return;
    }
    // release the resources
    finally { releaseResource(null, null, conn); }
} // end of createTable( )
```

The trouble is, after we run the page once, we get the following error because the table already exists.



So, we need to ask the database if a table by this name already exists and if so simply display a message.

First, at the top of the createTable( ) method add in two new variables.
A ResultSet to hold the reply from the database and a boolean flag so the program knows if a table already exists or not.

```
public void createTable(String tableName)
{
    Connection conn = null;
    ResultSet rs = null;
    boolean createTable = true;

    // Connect to MySQL
    try
    {
       conn = this.getConnection();
    }
    catch (SQLException e)
```

Right after the getConnection( ) call (inside of createTable( )) add this code inside the second try { } block.

```
// Connect to MySQL
try
{
   conn = this.getConnection();
}
catch (SQLException e)
{
    System.out.println("ERROR: Could not connect to the database");
    e.printStackTrace();
    return;
}

// Create a table
try
{
   // Check to see if a table already exists
   DatabaseMetaData meta = conn.getMetaData();
   rs = meta.getTables(null, null, "%", null);
   // Does the table already exist?
   // Loop through looking for the table name
   while (rs.next())
   {
      if(rs.getString(3).equals(tableName))
      {
         createTable=false;
         break;
      }
    }

   if(createTable)
   {
      String createString =
        "CREATE TABLE " + tableName + " ( " +
        "id INTEGER NOT NULL AUTO_INCREMENT, " +
        "name varchar(40) NOT NULL, " +
        "street varchar(40) NOT NULL, " +
```

And finally, at thee bottom of the method, in the finally { } block, add in the rs object so the resultset is released as well as the connection.

```
        return;
    }
    // release the resources
    finally { releaseResource(rs, null, conn); }
} // end of createTable( )
```

Now, when you run the program over and over you should see the following output:

INSERT a Record - Hard-Coded Data

We have our connection.
We have our table.
Now we want to add new records into the table.

Add a new method after main( ). You might want to organize the CRUD methods in the class alphabetically.

Type in this code:

```
public void insertData( )
{
   Connection conn = null;
   String sql = "";
   try
   {
      conn = this.getConnection();
   } catch (SQLException e) {
      System.out.println("ERROR: Could not connect to the database");
      e.printStackTrace();
   }

   // Insert the data
   try
   {
      sql = "INSERT INTO customer (name, street, city, state, postalcode)"
        + "VALUES ('Tom B. Erichsen','Skagen 21','Stavanger','MN','40069')";
      this.executeUpdate(conn, sql);
      System.out.println("Inserted a record with hard-coded data.");
    }
    catch (SQLException e)
    {
       System.out.println("ERROR: Could not insert the data using this SQL: " + sql);
       e.printStackTrace();
    }
   // Release the resources
   finally { releaseResource(null, null, conn); }
}// end of insertData( )
```

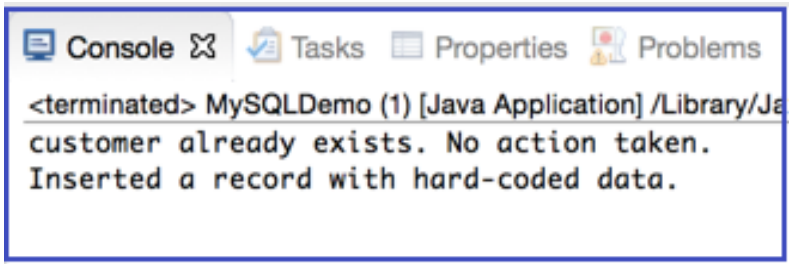
Don't forget to call the function in main( )!

Once again, we establish a connection with the database.
For the SQL we use the INSERT INTO statement hard-coding in the values.
This allows us to test the method and get a good reference on how the SQL must be written.

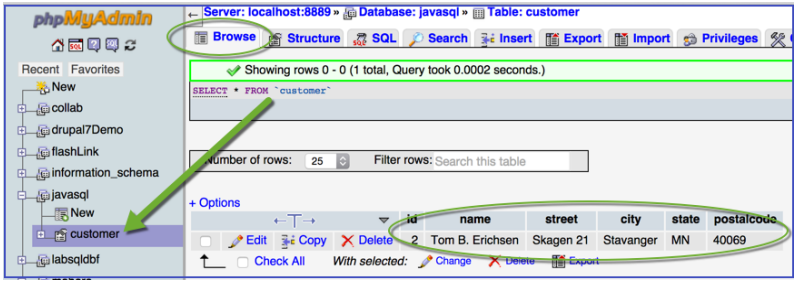
Some points to pay attention to:

1. If there is an error it will probably be in the SQL statement so we print out a copy of the sql string.
2. The SQL does not include an id field. That is because we set the id to auto-increment and the mySQL server will take care of this field automatically. (That just saved us from writing about 100+ lines of code.)
3. The data has single quotes around each item because in the database they are all listed as varchar and char which are Strings.
4. We are using hard-coded data to get the SQL string correct. Once that is good then we will replace each data item with a variable. This will help us get the double and single quotes and commas laid out correctly.

Here is what the results should look like:



And if you check the database table (http://localhost:8888/phpmyadmin) you should see the data. You may have to refresh the screen if it is already displaying.



INSERT a Record - Dynamic Data

Once you have the SQL INSERT INTO statement working you can replace the hard-coded values with dynamic data.

Inside the main( ) set up an array with the data you want to insert. This will be used to simulate actual input from a user.
Then, call the insertData( ) method passing in the String array as well as the name of the table.

Type in this code:

```
public static void main(String[] args)
{
   // Simulate data input by user, stored in an array
   String[ ] dataInput = {"Fred", "123 Test Avenue", "Two Egg", "FL", "12345"};
   MySQLDemo app = new MySQLDemo();

   app.createTable(TABLE_NAME);
   app.insertData(dataInput, TABLE_NAME);

} // end of main( )
```

Edit the insertData( ) method so it will accept the String array and table name keeping the original hard-coded SQL statement as a reference.

Make these changes to the insertData( ) method:

```
public void insertData(String[ ] dataArray, String thisTable)
{
   Connection conn = null;
   String sql = "";
   try
   {
      conn = this.getConnection();
   } catch (SQLException e) {
      System.out.println("ERROR: Could not connect to the database");
      e.printStackTrace();
   }

   // Insert the data
   try
   {
      /* REFERENCE SQL:
       *  sql = "INSERT INTO customer (name, street, city, state, postalcode)"
       *     + "VALUES ('Tom B. Erichsen','Skagen 21','Stavanger','MN','4006')";
       */
      sql = "INSERT INTO " + thisTable + " (name, street, city, state, postalcode) VALUES("
        + "'" + dataArray[0] + "', "
        + "'" + dataArray[1] + "', "
        + "'" + dataArray[2] + "', "
        + "'" + dataArray[3] + "', "
        + "'" + dataArray[4] + "')";
      this.executeUpdate(conn, sql);
      System.out.println("Inserted a record:" + dataArray[0] + " " + dataArray[1]);
    }
    catch (SQLException e)
    {
       System.out.println("ERROR: Could not insert the data using this SQL: " + sql);
       e.printStackTrace();
    }
   // Release the resources
   finally { releaseResource(null, null, conn); }
}// end of insertData( )
```

Replace hard-coded values with values from the array passed in as a parameter.

Get in the practice of formatting your SQL code in a specific pattern. This will help prevent mistakes such as missing characters. (The single quote and the commas are usually the culprits.)

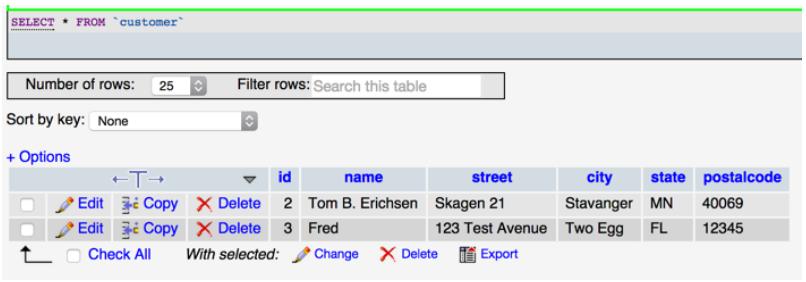
By lining up the + on each line any missing characters will quickly show themselves. Eclipse won't catch any errors because it only knows that you are typing a String and anything can go in a String!

You could have passed in individual values, but keep in mind that most tables have LOTS of fields and array is a great way to move all that data from one place to the next.

When you run the program now you should see these results:

```
customer already exists. No action taken.
Inserted a record:Fred 123 Test Avenue
```

And the data will begin to add up in the table:


SELECT * FROM `customer`
Number of rows: 25
Filter rows: Search this table
Sort by key: None
+ Options
id
name
street
city
state
postalcode
Edit Copy Delete 2 Tom B. Erichsen Skagen 21 Stavanger MN 40069
Edit Copy Delete 3 Fred 123 Test Avenue Two Egg FL 12345
Check All
With selected:
Change
Delete
Export

SELECT data - showTable( )

It sure is getting tiresome looking at the PHPMyAdmin page to watch the data grow.

I know…. let's write a showTable( ) method!

Here's the code:

```
public void showTable(String tableName)
{
   String sql = "";
   Statement stmt = null;
   ResultSet rs = null;
   int id = 0;
   String name = "";
   String street = "";
   String city = "";
   String state = "";
   String postalcode = "";

   // Connect to MySQL
   Connection conn = null;
   try {
      conn = this.getConnection();
   }
   catch (SQLException e)
   {
      System.out.println("ERROR: Could not connect to the database");
         e.printStackTrace();
   }

    // Select the data
    try
    {
       sql = "SELECT * FROM customer";
       // Run the SQL and save in Result Set
       stmt = conn.createStatement( );
       rs = stmt.executeQuery(sql);
       System.out.println("\nID\tNAME\t\tSTREET\t\t\tCITY - STATE - ZIP");
       System.out.println("*******************************************************************************");
       while (rs.next())
       {
          id = rs.getInt("id");
          name       = rs.getString("name");
          street     = rs.getString("street");
          city       = rs.getString("city");
          state      = rs.getString("state");
          postalcode = rs.getString("postalcode");
          System.out.printf("%d\t%s\t\t%s\t\t%s, %s %s.\n",
                id, name, street, city, state, postalcode);
        }
      }
      catch (SQLException e)
      {
         System.out.println("ERROR: Could not SELECT data using this SQL: " + sql);
            e.printStackTrace();
      }
      // Release the resources
      finally { releaseResource(rs, stmt, conn); }
} // end of showTable( )
```

1. At the top of the method set variables to hold the data from the table.

2. Open up a connection
3. Use the SQL SELECT command to extract all of the fields from the table name (passed in as a String parameter).
4. Use the executeQuery( ) to run the query that will return a result set (rs) …

Wait, we don't have a executeQuery( ) method!

Let's write one:

(You might want to put this down in the "heavy lifting" section of the class, right before the executeUpdate( ) method.)

```
/**
 * executeQuery - Run a SQL command which returns a result set:
 * SELECT
 *
 * @throws SQLException If something goes wrong
 * @return ResultSet containing data from the table
 */
public ResultSet executeQuery(Connection conn, String command) throws SQLException
{
   ResultSet rs;
   Statement stmt = null;
    try
    {
        stmt = conn.createStatement();
        rs = stmt.executeQuery(command); // This will throw a SQLException if it fails
        return rs;
    }
    finally
    {
        // This will run whether we throw an exception or not
        if (stmt != null) { stmt.close(); }
    }
} // end of executeQuery( )
```

rs - ResultSet

This is very similar to the executeUpdate( ) except that it returns a return set (rs).

Every ResultSet object has a getString( ) method that returns the contents of any field that is in the results.

For example, `name = rs.getString("name");`

Output a header for all the data.

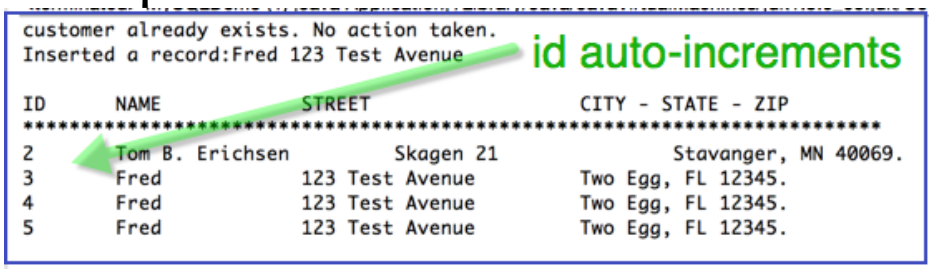
Then set up a while( ) loop that will walk through the resultset displaying a record at a time by transferring the data in the resultset to the appropriate variables. Use printf( ) to output the contents of that record.

Use \t "escape t" to space the columns to match the heading. This isn't a perfect solution because long data strings (see the first name in the output screenshot below) will through the column off but it is good enough for the console window. Later, when you use JavaFX this type of output problem will go away.

Add a call in the main( ) method and run the program.

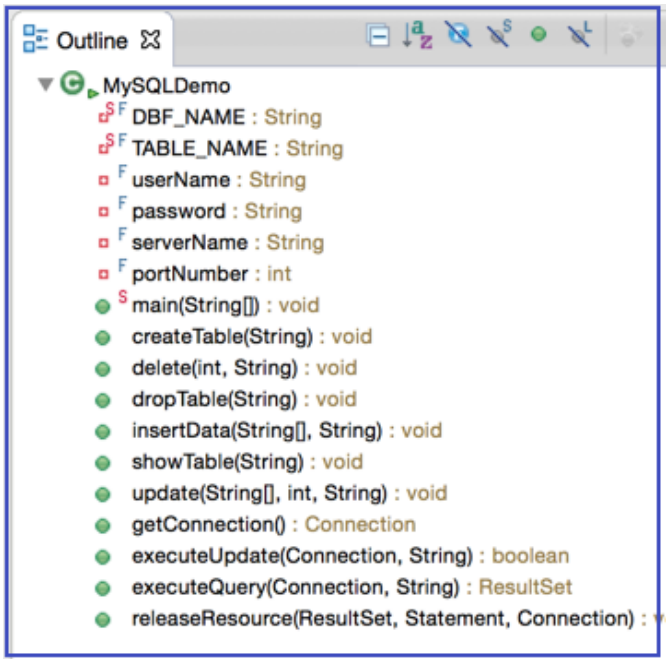
```
app.showTable(TABLE_NAME);
```

Your output should look similar to this:



The id field is set to auto-increment so the database automatically supplies the next highest number that is unique even though the rest of the data is the same. (Remember that each time you run the program

Programming tip: You are starting to build up quite a few methods in the class. Don't forget to use the Eclipse Outline View (upper right corner of the Java Perspective) in order to move from one function to another. It is so much faster than scroll, stop, read, scroll, stop, read, as you scan through your code looking for a particular method. (Some of the methods shown in this screen shot will be completed later in this tutorial.)


Outline
MySQLDemo
DBF_NAME : String
TABLE_NAME : String
userName : String
password : String
serverName : String
portNumber : int
main(String[]) : void
createTable(String) : void
delete(int, String) : void
dropTable(String) : void
insertData(String[], String) : void
showTable(String) : void
update(String[], int, String) : void
getConnection() : Connection
executeUpdate(Connection, String) : boolean
executeQuery(Connection, String) : ResultSet
releaseResource(ResultSet, Statement, Connection) :

UPDATE a Record - Hard-Coded Data

One of the elements of the CRUD model is UPDATE.
This SQL command allows you to change the data in any record.

Type in this code:

```
public void update()
{
   Connection conn = null;
   String sql = "";

   try
   {
      conn = this.getConnection();
   }
   catch (SQLException e)
   {
      System.out.println("ERROR: Could not connect to the database");
      e.printStackTrace();
   }

   // Update a record
   try
   {
       sql = "UPDATE Customer "
            + "SET name='Alicia', street='333 Happy Ave', "
            + "city='Bangor', state='ME', postalcode='12333' "
            + "WHERE id=5";
       this.executeUpdate(conn, sql);
    }
    catch (SQLException e)
    {
       System.out.println("ERROR: Could not update the record using this SQL: " + sql);
       e.printStackTrace();
    }
   // Release the resources
   finally { releaseResource(null, null, conn); }
}// end of insertData( )
```

1. Set up the connection.
2. Write the SQL (hard-coded to verify that the syntax is correct.)

   Make certain the id in the WHERE clause matches an id that you have in your own table!
3. Run executeUpdate( )
4. Release the resources.

Notice a pattern yet?

In the main call the update( ) method and showTable( ) a second time to

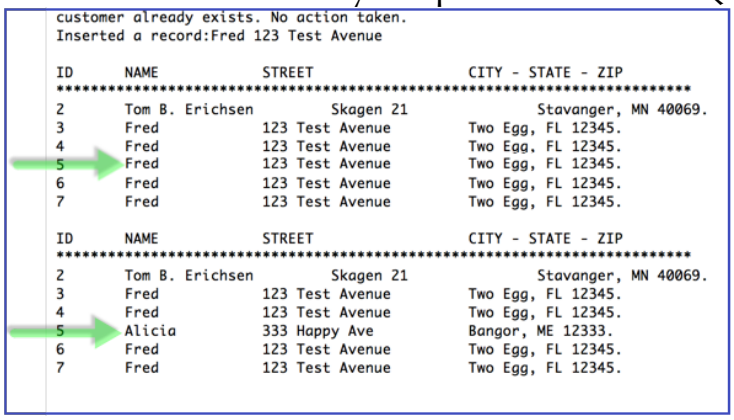
reveal the changes.

```
public static void main(String[] args)
{
   // Simulate data input by user, stored in an array
   String[ ] dataInput = {"Fred", "123 Test Avenue", "Two Egg", "FL", "12345"};
   MySQLDemo app = new MySQLDemo();

   app.createTable(TABLE_NAME);
   app.insertData(dataInput, TABLE_NAME);
   app.showTable(TABLE_NAME);
   app.update( );
   app.showTable(TABLE_NAME);

} // end of main( )
```

You should be able to see the change from the original data to the updated data for the record id you specified in the SQL statement.



```
customer already exists. No action taken.
Inserted a record:Fred 123 Test Avenue

ID      NAME            STREET                  CITY - STATE - ZIP
*************************************************************************
2       Tom B. Erichsen         Skagen 21               Stavanger, MN 40069.
3       Fred            123 Test Avenue         Two Egg, FL 12345.
4       Fred            123 Test Avenue         Two Egg, FL 12345.
5       Fred            123 Test Avenue         Two Egg, FL 12345.
6       Fred            123 Test Avenue         Two Egg, FL 12345.
7       Fred            123 Test Avenue         Two Egg, FL 12345.

ID      NAME            STREET                  CITY - STATE - ZIP
*************************************************************************
2       Tom B. Erichsen         Skagen 21               Stavanger, MN 40069.
3       Fred            123 Test Avenue         Two Egg, FL 12345.
4       Fred            123 Test Avenue         Two Egg, FL 12345.
5       Alicia          333 Happy Ave           Bangor, ME 12333.
6       Fred            123 Test Avenue         Two Egg, FL 12345.
7       Fred            123 Test Avenue         Two Egg, FL 12345.
```

UPDATE a Record - Dynamic Data

Now that you have a valid SQL statement, save it as a reference comment and replace each of the fields with a variable or, in this example, the element of an array that is passed into the method.

Make these changes to the update( ) method:

```
public void update(String[ ] dataArray, int thisID, String thisTable)
{
   Connection conn = null;
   String sql = "";

   try
   {
      conn = this.getConnection();
   }
   catch (SQLException e)
   {
      System.out.println("ERROR: Could not connect to the database");
      e.printStackTrace();
   }

   // Update a record
   try
   {
      /* REFERENCE SQL:
       * sql = "UPDATE Customer "
       *       + "SET name='Alicia', street='333 Happy Ave', "
       *       + "city='Bangor', state='ME', postalcode='12333' "
       *       + "WHERE id=30";
       */
       sql = "UPDATE " + thisTable
             + " SET name='"  + dataArray[0] + "', "
             + "street='"     + dataArray[1] + "', "
             + "city='"       + dataArray[2] + "', "
             + "state='"      + dataArray[3] + "', "
             + "postalcode='" + dataArray[4] + "' "
             + "WHERE id=" + thisID;
       this.executeUpdate(conn, sql);
    }
    catch (SQLException e)
    {
       System.out.println("ERROR: Could not update the record using this SQL: " + sql);
       e.printStackTrace();
    }
   // Release the resources
   finally { releaseResource(null, null, conn); }
}// end of insertData( )
```

The same code-style that we used with INSERT is useful here as well. By arranging the fields one to a line and aligning the + operators you are able to better able keep track of each double quote, single quote, space, and comma.

What data type is in the database table?

It is important to remember that each of the fields being updated is a String (varchar and char) in the database, so each item must have quotes around it. However, if any of the fields were numeric (int, float, double) then the single quotes would not be used.

In the main( ) method reuse the dataInput array by assigning new data to each index. This will simulate collecting information from a user-input form.
Call update( ) with the three parameters, the array containing the new values, the record id that should be updated, and the table name.

Here is a view of these changes:

```
public static void main(String[] args)
{
   // Simulate data input by user, stored in an array
   String[ ] dataInput = {"Fred", "123 Test Avenue", "Two Egg", "FL", "12345"};
   MySQLDemo app = new MySQLDemo();

   app.createTable(TABLE_NAME);
   app.insertData(dataInput, TABLE_NAME);
   app.showTable(TABLE_NAME);
   // Simulate new data input by user
   dataInput[0] = "Chris";
   dataInput[1] = "111 One Ave NE";
   dataInput[2] = "Princeton";
   dataInput[3] = "CA";
   dataInput[4] = "99987";
   app.update(dataInput, 7, TABLE_NAME);
   app.showTable(TABLE_NAME);
} // end of main( )
```

Using the same array, pass in new data to update( ).

The second parameter must be a record id that is in your database.

When you run the program you should see record id you designated has been changed.

```
customer already exists. No action taken.
Inserted a record:Fred 123 Test Avenue

ID      NAME            STREET                  CITY - STATE - ZIP
*********************************************************************
2       Tom B. Erichsen         Skagen 21               Stavanger, MN 40069.
3       Fred            123 Test Avenue         Two Egg, FL 12345.
4       Fred            123 Test Avenue         Two Egg, FL 12345.
5       Alicia          333 Happy Ave           Bangor, ME 12333.
6       Fred            123 Test Avenue         Two Egg, FL 12345.
7       Fred            123 Test Avenue         Two Egg, FL 12345.
8       Fred            123 Test Avenue         Two Egg, FL 12345.

ID      NAME            STREET                  CITY - STATE - ZIP
*********************************************************************
2       Tom B. Erichsen         Skagen 21               Stavanger, MN 40069.
3       Fred            123 Test Avenue         Two Egg, FL 12345.
4       Fred            123 Test Avenue         Two Egg, FL 12345.
5       Alicia          333 Happy Ave           Bangor, ME 12333.
6       Fred            123 Test Avenue         Two Egg, FL 12345.
7       Chris           111 One Ave NE          Princeton, CA 99987.
8       Fred            123 Test Avenue         Two Egg, FL 12345.
```

DELETE a record

Whoa! We are getting a lot of duplicate records here. Every time we run the program a new record is added.
Time to finish up our CRUD model with a delete( ) method.

Type in the code for delete( ):

```
public void delete(int thisID, String thisTable)
{
   Connection conn = null;
   String sql = "";

   try
   {
      conn = this.getConnection();
   } catch (SQLException e) {
      System.out.println("ERROR: Could not connect to the database");
      e.printStackTrace();
   }

   // Delete a record
   try
   {
      /* REFERENCE SQL:
       *  sql = "DELETE FROM customer WHERE id = 15";
       */
       sql = "DELETE FROM " + thisTable + " WHERE id = " + thisID;
       this.executeUpdate(conn, sql);
    }
    catch (SQLException e)
    {
       System.out.println("ERROR: Could delete the record using this SQL: " + sql);
       e.printStackTrace();
    }
   // Release the resources
   finally { releaseResource(null, null, conn); }
}// end of delete( )
```

Call the delete( ) method from inside of main( ) using a record id that is in your database table.
After the deletion is made show the table so you can see the results.

```
public static void main(String[] args)
{
    // Simulate data input by user, stored in an array
    String[ ] dataInput = {"Fred", "123 Test Avenue", "Two Egg", "FL", "12345"};
    MySQLDemo app = new MySQLDemo();

    app.createTable(TABLE_NAME);
    app.insertData(dataInput, TABLE_NAME);
    app.showTable(TABLE_NAME);
    // Simulate new data input by user
    dataInput[0] = "Chris";
    dataInput[1] = "111 One Ave NE";
    dataInput[2] = "Princeton";
    dataInput[3] = "CA";
    dataInput[4] = "99987";
    app.update(dataInput, 5    , TABLE_NAME);
    app.showTable(TABLE_NAME);
    app.delete(4,TABLE_NAME);
    app.showTable(TABLE_NAME);
} // end of main( )
```

Record id must match an id in your own table.

Your output should look similar to this:
(The updated data is the same because the code simply updated the same record with the same old information when the program was re-run again.)

```
Customer already exists. No action taken.
Inserted a record:Fred 123 Test Avenue
```

Original data

ID	NAME	STREET	CITY - STATE - ZIP
2	Tom B. Erichsen	Skagen 21	Stavanger, MN 40069.
3	Fred	123 Test Avenue	Two Egg, FL 12345.
4	Fred	123 Test Avenue	Two Egg, FL 12345.
5	Alicia	333 Happy Ave	Bangor, ME 12333.
6	Fred	123 Test Avenue	Two Egg, FL 12345.
7	Chris	111 One Ave NE	Princeton, CA 99987.
8	Fred	123 Test Avenue	Two Egg, FL 12345.
9	Fred	123 Test Avenue	Two Egg, FL 12345.

Updated Data

ID	NAME	STREET	CITY - STATE - ZIP
2	Tom B. Erichsen	Skagen 21	Stavanger, MN 40069.
3	Fred	123 Test Avenue	Two Egg, FL 12345.
4	Fred	123 Test Avenue	Two Egg, FL 12345.
5	Chris	111 One Ave NE	Princeton, CA 99987.
6	Fred	123 Test Avenue	Two Egg, FL 12345.
7	Chris	111 One Ave NE	Princeton, CA 99987.
8	Fred	123 Test Avenue	Two Egg, FL 12345.
9	Fred	123 Test Avenue	Two Egg, FL 12345.

Record Deleted

ID	NAME	STREET	CITY - STATE - ZIP
2	Tom B. Erichsen	Skagen 21	Stavanger, MN 40069.
3	Fred	123 Test Avenue	Two Egg, FL 12345.
5	Chris	111 One Ave NE	Princeton, CA 99987.
6	Fred	123 Test Avenue	Two Egg, FL 12345.
7	Chris	111 One Ave NE	Princeton, CA 99987.
8	Fred	123 Test Avenue	Two Egg, FL 12345.
9	Fred	123 Test Avenue	Two Egg, FL 12345.

DROP a table

We now have everything we need to manipulate the data in our database.

However, it may be necessary in the future to DROP or delete a table. For example, you may create a temporary table on the fly in your program that you need to remove before the program ends.

Just for grins and giggles (and because the Law of Programming dictates that for every create( ) there should be a delete( )) and write a delete( ) method.

Type in this code:

```
public void dropTable(String tableName)
{
   String sql = "";
   Connection conn = null;
   try
   {
      conn = this.getConnection();
   }
   catch (SQLException e)
   {
      System.out.println("ERROR: Could not connect to the database");
      e.printStackTrace();
   }
   try
   {
      sql = "DROP TABLE " + tableName;
      this.executeUpdate(conn, sql);
      System.out.println("Dropped the table named:" + tableName);
   }
   catch (SQLException e)
   {
      System.out.println("ERROR: Could not drop the table using this SQL: " + sql);
      e.printStackTrace();
      return;
  }
   finally { releaseResource(null,null, conn);}
} // end of dropTable( )
```

Same process:

1. Connect to the database
2. Write the SQL table, using the variable, tableName, passed in as a parameter.
3. Call executeUpdate( ) because no ResultSet will be returned.
4. Release the resources.

In main( ) add a test line of code that will drop the table:

```
public static void main(String[] args)
{
    // Simulate data input by user, stored in an array
    String[ ] dataInput = {"Fred", "123 Test Avenue", "Two Egg", "FL", "12345"};
    MySQLDemo app = new MySQLDemo();

    app.createTable(TABLE_NAME);
    app.insertData(dataInput, TABLE_NAME);
    app.showTable(TABLE_NAME);
    // Simulate new data input by user
    dataInput[0] = "Chris";
    dataInput[1] = "111 One Ave NE";
    dataInput[2] = "Princeton";
    dataInput[3] = "CA";
    dataInput[4] = "99987";
    app.update(dataInput, 5    , TABLE_NAME);
    app.showTable(TABLE_NAME);
    app.delete(4,TABLE_NAME);
    app.showTable(TABLE_NAME);

    app.dropTable(TABLE_NAME);
} // end of main( )
```

Here is the output on the first run:

```
customer already exists. No action taken.
Inserted a record:Fred 123 Test Avenue

ID      NAME            STREET                  CITY - STATE - ZIP
**************************************************************************
2       Tom B. Erichsen         Skagen 21               Stavanger, MN 40069.
3       Fred            123 Test Avenue         Two Egg, FL 12345.
5       Chris           111 One Ave NE          Princeton, CA 99987.
6       Fred            123 Test Avenue         Two Egg, FL 12345.
7       Chris           111 One Ave NE          Princeton, CA 99987.
8       Fred            123 Test Avenue         Two Egg, FL 12345.
9       Fred            123 Test Avenue         Two Egg, FL 12345.
10      Fred            123 Test Avenue         Two Egg, FL 12345.

ID      NAME            STREET                  CITY - STATE - ZIP
**************************************************************************
2       Tom B. Erichsen         Skagen 21               Stavanger, MN 40069.
3       Fred            123 Test Avenue         Two Egg, FL 12345.
5       Chris           111 One Ave NE          Princeton, CA 99987.
6       Fred            123 Test Avenue         Two Egg, FL 12345.
7       Chris           111 One Ave NE          Princeton, CA 99987.
8       Fred            123 Test Avenue         Two Egg, FL 12345.
9       Fred            123 Test Avenue         Two Egg, FL 12345.
10      Fred            123 Test Avenue         Two Egg, FL 12345.

ID      NAME            STREET                  CITY - STATE - ZIP
**************************************************************************
2       Tom B. Erichsen         Skagen 21               Stavanger, MN 40069.
3       Fred            123 Test Avenue         Two Egg, FL 12345.
5       Chris           111 One Ave NE          Princeton, CA 99987.
6       Fred            123 Test Avenue         Two Egg, FL 12345.
7       Chris           111 One Ave NE          Princeton, CA 99987.
8       Fred            123 Test Avenue         Two Egg, FL 12345.
9       Fred            123 Test Avenue         Two Egg, FL 12345.
10      Fred            123 Test Avenue         Two Egg, FL 12345.
Dropped the table named:customer
```

And if you re-run the program you will see that a new table was built and a single record added.

The update( ) and delete( ) methods don't do anything because they are set to id numbers that no longer exist.

```
Created table named:customer
Inserted a record:Fred 123 Test Avenue

ID      NAME            STREET                  CITY - STATE - ZIP
**************************************************************************
1       Fred            123 Test Avenue         Two Egg, FL 12345.

ID      NAME            STREET                  CITY - STATE - ZIP
**************************************************************************
1       Fred            123 Test Avenue         Two Egg, FL 12345.

ID      NAME            STREET                  CITY - STATE - ZIP
**************************************************************************
1       Fred            123 Test Avenue         Two Egg, FL 12345.
Dropped the table named:customer
```

New table built!

Inserted record

No matching id to update

No matching id to delete

Summary

With the techniques presented in this tutorial you now have the basic skills necessary to do CRUD on any relational database table.

You now have the code to:

- CREATE TABLE - Create an new, empty table as well as check if a table exists already.
- INSERT a new record of data into a table using dynamic data.
- SELECT a set of data from a table.
- UPDATE any record in a table using dynamic data.
- DELETE a record from a table.
- DROP (delete) a table.

You understand what CRUD stands for:

- Create - INSERT
- Read - SELECT
- Update - UPDATE
- Delete - DELETE

You should also appreciate the heavy-lifting power offered by these three methods:

DriveManager.getConnection( ) - Connect your Java program to a database. We will be using mySQL, but with a small edit and the correct software driver you can connect to any relational database.

Statement.executeUpdate( ) - Run SQL commands that manipulate the database tables but don't return any data. This would include CREATE, DROP, DELETE, UPDATE, and INSERT.

Statement.executeQuery( ) - Run SQL commands that return a result set. This would include SELECT statements.

Source Code

The complete source code for this project can be downloaded from Web Explorations: http://webexplorations.com/book/java/MySQLDemo.java.zip

```
/**
 * <strong>MySQLDemo.java</strong> - description
 *
 *          Based on original code from: http://www.ccs.neu.edu/
home/kathleen/classes/cs3200/JDBCtutorial.pdf
 * @author Peter K. Johnson - <a href="http://
WebExplorations.com"
 *          target="_blank"> http://WebExplorations.com</a><br >
 *          Written: Oct 27, 2014<br >
 *          Revised: Oct 31, 2014
 */
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

/**
 * This class demonstrates how to connect to MySQL and run some
basic commands.
 *
 * In order to use this, you have to download the Connector/J
driver and add
 * its .jar file to your build path.  You can find it here:
 * http://dev.mysql.com/downloads/connector/j/
 *
 * ERROR MESSAGES?
 * ERROR: Could not connect to the
databasecom.mysql.jdbc.exceptions.jdbc4.
 * CommunicationsException: Communications link failure
```

```
 * Solution: Make sure MAMP/WAMP is running ;-)
 *
 * ERROR: java.sql.SQLException: No suitable driver found for
jdbc:mysql://localhost:3306/
 * The JDBC Driver is not on the path (Eclipse is not aware of
it.)
 * To add it to your class path:
 * 1. Right click on your project
 * 2. Go to Build Path -> Add External Archives...
 * 3. Select the file mysql-connector-java-5.1.24-bin.jar
 *    NOTE: If you have a different version of the .jar file,
the name may be a little different.
 *
 * ERROR: java.sql.SQLException: Access denied for user
'userName'@'localhost' (using password: YES)
 * The userID and password are wrong.
 * The user name and password are both "root" or "root" ""
depending on which LAMP stack you are running.
 * You can change these setting below.
 */
public class MySQLDemo
{
    // Set up two constants to be used in this test environment.
    private final static String DBF_NAME = "javasql";
    private final static String TABLE_NAME = "customer";

    // The mySQL username
    private final String userName = "root";

    // The mySQL password (may be empty "" )
    private final String password = "root";

    // Name of the computer running mySQL
    private final String serverName = "localhost";

    // Port of the MySQL server (default is 3306 or 8889 on
MAMP)
    private final int portNumber = 8889;
```

```
    public static void main(String[] args)
    {
        // Simulate data input by user, stored in an array
        String[ ] dataInput = {"Fred", "123 Test Avenue", "Two
Egg", "FL", "12345"};
        MySQLDemo app = new MySQLDemo();

        app.createTable(TABLE_NAME);
        app.insertData(dataInput, TABLE_NAME);
        app.showTable(TABLE_NAME);
        // Simulate new data input by user
        dataInput[0] = "Chris";
        dataInput[1] = "111 One Ave NE";
        dataInput[2] = "Princeton";
        dataInput[3] = "CA";
        dataInput[4] = "99987";
        app.update(dataInput, 5    , TABLE_NAME);
        app.showTable(TABLE_NAME);
        app.delete(4,TABLE_NAME);
        app.showTable(TABLE_NAME);

        app.dropTable(TABLE_NAME);
    } // end of main( )

    /**
     * createTable - create a new table in the database
     *               if one already exists no action is taken
     * @param tableName
     */
    public void createTable(String tableName)
    {
        Connection conn = null;
        String sql = "";
        ResultSet rs = null;
        boolean createTable = true;
```

```
        // Connect to MySQL
        try
        {
           conn = this.getConnection();
        }
        catch (SQLException e)
        {
            System.out.println("ERROR: Could not connect to the
database");
            e.printStackTrace();
            return;
        }

        // Create a table
        try
        {
           // Check to see if a table already exists
           DatabaseMetaData meta = conn.getMetaData();
           rs = meta.getTables(null, null, "%", null);
           // Does the table already exist?
           // Loop through looking for the table name
           while (rs.next())
           {
              if(rs.getString(3).equals(tableName))
              {
                 createTable=false;
                 break;
              }
            }

           if(createTable)
           {
              String createString =
                "CREATE TABLE " + tableName + " ( " +
                "id INTEGER NOT NULL AUTO_INCREMENT, " +
                "name varchar(40) NOT NULL, " +
                "street varchar(40) NOT NULL, " +
                "city varchar(20) NOT NULL, " +
```

```
                "state char(2) NOT NULL, " +
                "postalcode char(5), " +
                "PRIMARY KEY (id))";
              this.executeUpdate(conn, createString);
              System.out.println("Created table named:" +
tableName);
            }
            else // table already exists
            {
               System.out.println(tableName + " already exists.
No action taken.");
            }
        }
        catch (SQLException e)
        {
            System.out.println("ERROR: Could not create the
table named: " + tableName);
            e.printStackTrace();
            return;
        }
        // release the resources
        finally { releaseResource(rs, null, conn); }
    } // end of createTable( )


    /**
     * delete( ) - remove a record based on id
     * @param thisID - the id of the record to be removed
     * @param thisTable
     */
    public void delete(int thisID, String thisTable)
    {
       Connection conn = null;
       String sql = "";

       try
       {
          conn = this.getConnection();
```

```
        } catch (SQLException e) {
           System.out.println("ERROR: Could not connect to the
database");
           e.printStackTrace();
        }

        // Delete a record
        try
        {
           /* REFERENCE SQL:
            *  sql = "DELETE FROM customer WHERE id = 15";
            */
            sql = "DELETE FROM " + thisTable + " WHERE id = " +
thisID;
            this.executeUpdate(conn, sql);
         }
         catch (SQLException e)
         {
            System.out.println("ERROR: Could delete the record
using this SQL: " + sql);
            e.printStackTrace();
         }
        // Release the resources
        finally { releaseResource(null, null, conn); }
    }// end of delete( )

    /**
     * dropTable - removes the specific table from the database
     * @param tableName
     */
    public void dropTable(String tableName)
    {
        String sql = "";
        Connection conn = null;
        try
        {
           conn = this.getConnection();
        }
```

```
        catch (SQLException e)
        {
           System.out.println("ERROR: Could not connect to the
database");
           e.printStackTrace();
        }
        try
        {
           sql = "DROP TABLE " + tableName;
           this.executeUpdate(conn, sql);
           System.out.println("Dropped the table named:" +
tableName);
        }
        catch (SQLException e)
        {
           System.out.println("ERROR: Could not drop the table
using this SQL: " + sql);
           e.printStackTrace();
           return;
       }
        finally { releaseResource(null,null, conn);}
    } // end of dropTable( )


    /**
     * insertData - inserts data from the array into the
designated table
     * @param dataArray  - Must be in the indexed order:<br>
     *                     0-name 1-street 2-city 3-state 4-
postalcode
     * @param thisTable
     */
    public void insertData(String[ ] dataArray, String
thisTable)
    {
       Connection conn = null;
       String sql = "";
       try
```

```
        {
           conn = this.getConnection();
        } catch (SQLException e) {
           System.out.println("ERROR: Could not connect to the
database");
           e.printStackTrace();
        }

        // Insert the data
        try
        {
           /* REFERENCE SQL:
            *  sql = "INSERT INTO customer (name, street, city,
state, postalcode)"
            *    + "VALUES ('Tom B. Erichsen','Skagen
21','Stavanger','MN','4006')";
            */
            sql = "INSERT INTO " + thisTable + " (name, street,
city, state, postalcode) VALUES("
              + "'" + dataArray[0] + "', "
              + "'" + dataArray[1] + "', "
              + "'" + dataArray[2] + "', "
              + "'" + dataArray[3] + "', "
              + "'" + dataArray[4] + "')";
            this.executeUpdate(conn, sql);
            System.out.println("Inserted a record:" +
dataArray[0] + " " + dataArray[1]);
         }
         catch (SQLException e)
         {
            System.out.println("ERROR: Could not insert the data
using this SQL: " + sql);
            e.printStackTrace();
         }
        // Release the resources
        finally { releaseResource(null, null, conn); }
    }// end of insertData( )
```

```
    /**
     * showTable( ) - display the contents of the designated
table
     * @param tableName
     */
    public void showTable(String tableName)
    {
       String sql = "";
       Statement stmt = null;
       ResultSet rs = null;
       int id = 0;
       String name = "";
       String street = "";
       String city = "";
       String state = "";
       String postalcode = "";

       // Connect to MySQL
       Connection conn = null;
       try {
          conn = this.getConnection();
       }
       catch (SQLException e)
       {
          System.out.println("ERROR: Could not connect to the
database");
             e.printStackTrace();
       }

        // Select the data
        try
        {
           sql = "SELECT * FROM customer";
           // Run the SQL and save in Result Set
           stmt = conn.createStatement( );
           rs = stmt.executeQuery(sql);
           System.out.println("\nID\tNAME\t\tSTREET\t\t\tCITY -
```

```
STATE - ZIP");

System.out.println("*********************************************
*****************************");
           while (rs.next())
           {
              id = rs.getInt("id");
              name       = rs.getString("name");
              street     = rs.getString("street");
              city       = rs.getString("city");
              state      = rs.getString("state");
              postalcode = rs.getString("postalcode");
              System.out.printf("%d\t%s\t\t%s\t\t%s, %s %s.\n",
                    id, name, street, city, state, postalcode);
            }
          }
          catch (SQLException e)
          {
             System.out.println("ERROR: Could not SELECT data
using this SQL: " + sql);
                e.printStackTrace();
          }
          // Release the resources
          finally { releaseResource(rs, stmt, conn); }
    } // end of showTable( )


    /**
     * update( ) - update a specific record using the contents
of the array based on a specific ID
     * @param dataArray Must be in the indexed order:<br>
     *                  0-name 1-street 2-city 3-state 4-
postalcode
     * @param thisID
     * @param thisTable
     */
    public void update(String[ ] dataArray, int thisID, String
thisTable)
```

```
    {
       Connection conn = null;
       String sql = "";

       try
       {
          conn = this.getConnection();
       }
       catch (SQLException e)
       {
          System.out.println("ERROR: Could not connect to the
database");
          e.printStackTrace();
       }

       // Update a record
       try
       {
          /* REFERENCE SQL:
           * sql = "UPDATE Customer "
           *      + "SET name='Alicia', street='333 Happy Ave',
"
           *      + "city='Bangor', state='ME',
postalcode='12333' "
           *      + "WHERE id=30";
           */
           sql = "UPDATE " + thisTable
                 + " SET name='"  + dataArray[0] + "', "
                 + "street='"     + dataArray[1] + "', "
                 + "city='"       + dataArray[2] + "', "
                 + "state='"      + dataArray[3] + "', "
                 + "postalcode='" + dataArray[4] + "' "
                 + "WHERE id=" + thisID;
           this.executeUpdate(conn, sql);
        }
        catch (SQLException e)
        {
           System.out.println("ERROR: Could not update the
```

```
record using this SQL: " + sql);
           e.printStackTrace();
        }
       // Release the resources
       finally { releaseResource(null, null, conn); }
    }// end of insertData( )


    /* ******************************
     * HEAVY LIFTING - Common to the CRUD methods.
     ********************************/
     /**
     * getConnection( ) - Get a new database connection
     *
     * @return Connection
     * @throws SQLException
     */
     public Connection getConnection() throws SQLException
     {
        Connection conn = null;
        Properties connectionProps = new Properties();
        connectionProps.put("user", this.userName);
        connectionProps.put("password", this.password);

        conn = DriverManager.getConnection("jdbc:mysql://"
                + this.serverName + ":" + this.portNumber + "/"
+ DBF_NAME,
                connectionProps);
        return conn;
     }


    /**
     * executeUpdate( ) - Used to run a SQL command which does
NOT return a resultSet:
     * CREATE/INSERT/UPDATE/DELETE/DROP
     *
     * @throws SQLException If something goes wrong
```

```
     * @return boolean if command was successful or not
     */
     public boolean executeUpdate(Connection conn, String
command) throws SQLException
     {
        Statement stmt = null;
        try
        {
            stmt = conn.createStatement();
            stmt.executeUpdate(command); // This will throw a
SQLException if it fails
            return true;
        }
        finally
        {
            // This will run whether we throw an exception or
not
            if (stmt != null) { stmt.close(); }
        }
    } // end of executeUpdate( )


    /**
     * executeQuery - Run a SQL command which returns a result
set:
     * SELECT
     *
     * @throws SQLException If something goes wrong
     * @return ResultSet containing data from the table
     */
    public ResultSet executeQuery(Connection conn, String
command) throws SQLException
    {
       ResultSet rs;
       Statement stmt = null;
        try
        {
            stmt = conn.createStatement();
```

```
            rs = stmt.executeQuery(command); // This will throw
a SQLException if it fails
            return rs;
        }
        finally
        {
            // This will run whether we throw an exception or
not
            if (stmt != null) { stmt.close(); }
        }
    } // end of executeQuery( )


    /**
     * releaseResource( ) - Free up the system resources that
were opened.
     *                      If not used,  a null will be passed
in for that parameter.
     * @param rs - Resultset
     * @param ps - Statement
     * @param conn - Connection
     */
    public void releaseResource(ResultSet rs, Statement ps,
Connection conn )
    {
       if (rs != null)
       {
          try { rs.close(); }
          catch (SQLException e) { /* ignored */}
       }
       if (ps != null)
       {
          try { ps.close(); }
          catch (SQLException e) { /* ignored */}
       }
       if (conn != null)
       {
          try { conn.close();}
```

```
            catch (SQLException e) { /* ignored */}
        }
    } // end of releaseResource( )
}
```

Credits

Tutorial: Java SQL written by Peter K. Johnson, Web Explorations, LLC

Last update: 11/01/2014