CS108, Stanford
Winter 2013

Handout #23
Young

JDBC

JDBC allows us to connect our Java programs to a database.

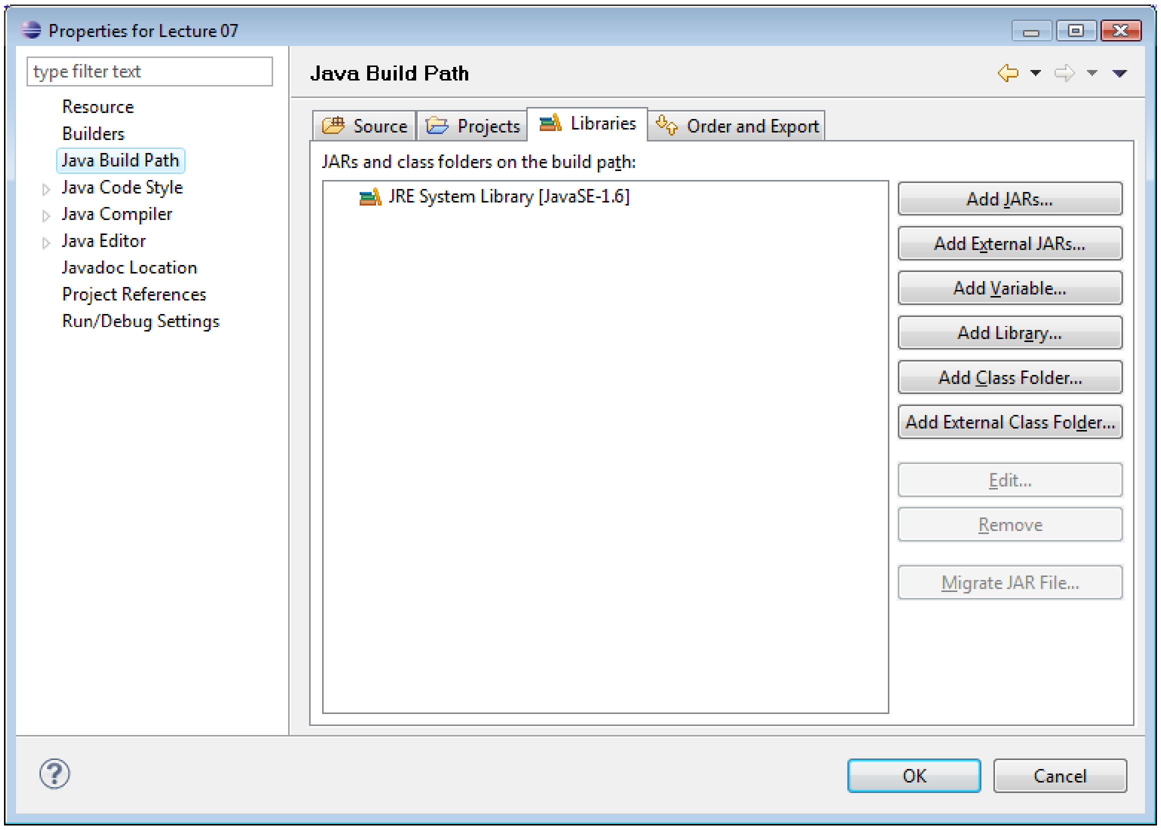
The MySQL JDBC Driver

JDBC can work with a variety of different types of databases. A JDBC Driver connects Java to a particular type of database. In our case, we'll need a JDBC driver for MySQL. We will use the Connector/J driver from MySQL.com available at:

`http://www.mysql.com/products/connector/`

Choose the "Platform Independent" option and download the driver. The download will be a zip file including Java source code and a JAR file with the source pre-compiled. Ignore the source code, we are only interested in the JAR file. Go ahead and store the `mysql-connector-java-5.1.33-bin.jar` file somewhere handy.

When creating a project which uses JDBC you'll need to tell Eclipse where your JDBC driver is. You can do this by right-clicking on the project folder in Eclipse and select "Properties". This will bring up the project's properties window, switch to the "Java Build Path" section and select the "Libraries" tab:



JARs and class folders on the build path:

JRE System Library [JavaSE-1.6]

Click on the "Add External JARs" button on the right side of the window. Find and add the `mysql-connector-java-5.1.28-bin.jar` file.

A Simple JDBC Program

When working with JDBC we'll need the same pieces of information we needed when working with MySQL directly. Make sure you have the following information which you should have received via e-mail:

- Your MySQL Username
- Your MySQL Initial Password
- The name of the Stanford MySQL Database Server
- Your MySQL Database Name

To make grading easier for the TAs, we've created a special Java file you will use to store this information called MyDBInfo.java. Here's what it looks like:

```
/*
 * CS108 Student: This file will be replaced when we test
 * your code. So, do not add any of your
 * assignment code to this file. Also, do not modify
 * the public interface of this file.
 * Only change the public MyDBInfo constants so that it
 * works with the database login credentials
 * that we emailed to you.
 */
public class MyDBInfo {

   public static final String MYSQL_USERNAME = "ccs108yourStudentID";
   public static final String MYSQL_PASSWORD = "yourpassword";
   public static final String MYSQL_DATABASE_SERVER
                                  = "mysql-user.stanford.edu";
   public static final String MYSQL_DATABASE_NAME = "c_cs108_yourStudentID";

}
```

As you can see it consists of a number of public static Strings which correspond to your username, password, database name, and the database server. Replace the strings in this file with your actual data. Notice that your password is directly integrated into your Java program, which is why you should not be using your regular SUNetID password as your MySQL password.

In the previous handout on MySQL we created a table of metropolises and stored it in our MySQL database. We will now take a look at a program which accesses that table. Just as a reminder, here is what the table looked like:

```
+---------------+---------------+------------+
| metropolis    | continent     | population |
+---------------+---------------+------------+
| Mumbai        | Asia          | 20400000   |
| New York      | North America | 21295000   |
| San Francisco | North America | 5780000    |
| London        | Europe        | 8580000    |
| Rome          | Europe        | 2715000    |
| Melbourne     | Australia     | 3900000    |
| San Jose      | North America | 7354555    |
| Rostov-on-Don | Europe        | 1052000    |
+---------------+---------------+------------+
```

Here is a program which prints the metropolis and population data from the table:

```java
import java.sql.*;

public class ExampleAccess {

   static String account = MyDBInfo.MYSQL_USERNAME;
   static String password = MyDBInfo.MYSQL_PASSWORD;
   static String server = MyDBInfo.MYSQL_DATABASE_SERVER;
   static String database = MyDBInfo.MYSQL_DATABASE_NAME;

   public static void main(String[] args) {

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

         Connection con = DriverManager.getConnection
            ( "jdbc:mysql://" + server, account ,password);

         Statement stmt = con.createStatement();
         stmt.executeQuery("USE " + database);
         ResultSet rs = stmt.executeQuery("SELECT * FROM metropolises");

         while(rs.next()) {
            String name = rs.getString("metropolis");
            long pop = rs.getLong("population");
            System.out.println(name + "\t" + pop);
         }

         con.close();
      } catch (SQLException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
      catch (ClassNotFoundException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
   }

}
```

Okay, let's go ahead and break down what this program is doing. First notice that we need to import from `java.sql.*`.

Next as you can see, we go ahead and create some variables to keep track of your account, password, etc drawing the values from the MyDBInfo file we saw a moment ago.

We now need to make sure that our JDBC driver is loaded. That's done with the line:

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

This line makes sure that Java knows about the driver. It must be placed within a try catch block. You can see the corresponding exception which needs to be caught, ClassNotFoundException, at the end of the program.

Our next step is to create a connection between our Java program and the database. We do this by creating a Connection object:

```
    Connection con = DriverManager.getConnection(
           "jdbc:mysql://" + server, account ,password);
```

We use the DriverManager class (which only contains static methods). Its getConnection method needs the URL of the server, along with your account name and password. The URL should look something like this:

```
jdbc:mysql://mysql-user.stanford.edu
```

This method and subsequent SQL-related methods may throw SQL Exceptions and must be placed within a try catch block with a catch for SQLException.

Once we have our connection setup, we create a Statement object which can be used to execute standard SQL statements.

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

We follow up with a SQL statement to set the current database we are working with, and then do a SELECT on the metropolises table within the database.

```
stmt.executeQuery("USE " + database);
ResultSet rs = stmt.executeQuery("SELECT * FROM metropolises");
```

The results of the SELECT are stored into a ResultSet object. We can use the ResultSet object to go through the rows in the table returned by the SELECT. The `next` method returns true if more result rows exist, and it moves the current row forward. You can see we access individual data items within the row using getString and getInt passing in the name of the column we are interested in:

```
while(rs.next()) {
      String name = rs.getString("metropolis");
      int pop = rs.getInt("population");
      System.out.println(name + "\t" + pop);
}
```

Working with the Result Set

JDBC provides a number of different methods which allow us to work with the Result Set. The Result Set allows access to a single row from the results at a time. Let's take a look at some of the more important methods available. Keep in mind that complete documentation check is available in the online JavaDocs – ResultSet is in the java.sql.* package. All of the methods described have the potential to throw a SQLException.

Working with Rows

The current row is called "the cursor." We can move the cursor using a variety of different methods include:

- next() and previous() as you might expect move the cursor backwards and forward. next returns false if the cursor has moved beyond the last row in the result set. previous returns false if the cursor has moved beyond the first set.

- first() and last() move the cursor to the beginning and end respectively of the result set. They both return false if there are no rows in the result set.

- absolute(*x*) moves to a specific row within the set. Set rows are numbered starting with row 1 (i.e., indexing is <u>not</u> zero-based).

- relative(*x*) moves forward or backward relative to the current cursor position.

- beforeFirst() moves to just before the first row. The can be combined with next to get to the first row:

  ```
  ResultSet rs = …;

  rs.beforeFirst();
  rs.next();  // this moves us to the first item
  ```

- afterLast() moves to just after the last row and can be used with prev to get the last row:

```
ResultSet rs = …;

rs.afterLast();
rs.previous();  // this moves us to the last item
```

- getRow() returns the current row number. The following code can be used to get the number of rows:

  ```
  ResultSet rs = …;

  rs.last();
  int rowCount = rs.getRow();
  ```

Working with Columns

Once we move the cursor to the particular row we are interested in, we'll want to get data from a particular column. JDBC provides a variety of different get methods which retrieve the data in different data formats. Here are just a few: getString, getDate, getDouble, getInt. In general JDBC will perform type conversion from the information in the Result Set to whichever type you are requesting. Each get method has two different versions. One method takes an index number as a parameter the other takes a column name. Index numbers start with 1. Using our metropolises table as an example again, the following are equivalent:

```
ResultSet rs = …;
rs.getString("continent");
```

and

```
ResultSet rs = …;
rs.getString(2);
```

as Continent is the second column in our table.

Updating vs. Querying

The executeQuery method shown previously can only be used for querying. Methods which modify the database should use the executeUpdate method. For example the following line will delete metropolises from "North America":

```
stmt.executeUpdate(
  "DELETE FROM metropolises WHERE continent = \"North America\"");
```

Use executeUpdate for UPDATE, INSERT, and DELETE MySQL statements.

Closing the Connection

When you're all done with the database, please close the connection.

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

If you skip this step, the database connection will close after a set timeout interval completes. However the Stanford server only allows a limited number of collections, and you and your fellow classmates may get locked out if you open connections faster than they can timeout.