Basics of Using JDBC with MySQL in the LIDA

Prof. Dorothee Koch, Hochschule für Technik Stuttgart

The following chapters use some simple examples to explain how to access SQL databases from Java code by using a JDBC driver. Please be aware that version numbers change often, so the mentioned version numbers of some of the involved software may not be current.
One complete example has been prepared for you. You can start with executing it to get an idea of how it works.

1 Setting up JDBC with MySQL in the LIDA

- Log in to the server (193.196.143.2).
- Open a shell window (terminal).

With Netbeans

If you want to use Netbeans for developing the programs, execute the following steps:

Type into the shell window:
`module load ide/netbeans` (then hit return) followed by:
`netbeans`

If you prefer Eclipse:
`module load ide/eclipse`
`eclipse`

Alternatively, you can enter directly the path of the commands:
`/opt/hft/ide/netbeans/8.2/bin/netbeans`
or
`/opt/hft/ide/eclipse/neon/neon/eclipse/eclipse`

- Download the jar file containing the JDBC driver for MySQL from the Moodle.
- Start Netbeans (if entering `netbeans` to the shell prompt does not work, enter instead
  `/opt/hft/ide/netbeans/8.2/bin/netbeans`
  You can add the directory `/opt/hft/ide/netbeans/8.2/bin` to the path variable in your `.bashrc` so you don't need to enter the full path again).
- Select "Window" in the main menue and select "Services"

- In the Services menue that appears on the left, click on "Databases", then right click on "Drivers" => new Drivers => Add: Then add the path of the folder where you downloaded the JDBC driver. (the jar file). The driver now should appear within Netbeans in a list of drivers, for instance as MySQL(Connector/JDriver)
  Next, rightclick on this MySQL(Connector/JDriver).
- Chose "connect using" which makes a pop up window appear.
- In this window you need to enter:
  Host: 193.196.143.168
  Port: 3306
  Database: koch_universitydb
  User Name: your MySQL user name
  Password: your MySQL password
  JDBC URL: jdbc:mysql://193.196.143.168/koch_universitydb (this should appear automatically). In case there are any characters displayed after "koch_universitydb" (like for instance '?zeroDateTimeBehavior=...'), they should be erased.
- Click on "Test Connection" to test if it works.
- Then click on Finish.

- Chose in the File menue: "New Project".
- Select "Java Application" and click on "Next".
- A window appears in which you can chose a project name, for instance "dbtest". Uncheck the boxes named "create main class".
- Then click Finish.

- Copy the Java-Files from the Exercises directory in the Moodle into a shell window into your directory `~/NetBeansProjects/dbtest/src` (if dbtest is the project name you had chosen): Open the files in the Moodle and right click on the names to specify this folder. Check that the files are in your Netbeansfolder by selecting "Projects" and clicking on your project name. The files should now be there.

- Now set the Java library:
  Rightclick on your project name (dbtest). Select "Properties" (last in the pop-up list). Click on "Libraries", and then click on "Add JAR/Folder".
  For "Dateiname" (filename) enter the path where you saved your jar file (mysql-connector-java-8.0.23.jar)
- Click OK, and again OK.

Working in a Shell

Instead of using Netbeans, you can develop your Java classes, compile them and test the connection to the database simply by using a shell. This also works if you are working from home under Windows (without the X Window System)[1] and cannot open the Netbeans on the LIDA server. In order to do this, execute the following:

- In a shell window on the LIDA server, make a folder with a name of your choice (for instance JDBCTest)

1. Of course, you can install X Windows on your Windows computer, for instance cygwin. The you can run Netbeans on the LIDA server remotely from your home system.

- Copy the JAR file from the Moodle
  `mysql-connector-java-8.0.23.jar` into it. It contains the JDBC driver.
  This file is can be found in our Moodle course.
- Copy the files `JDBCTestExample.java` and `KeyboardInput.java` into the same folder.
  `KeyboardInput.java` is a class with methods for handling input from the keyboard. `JDBCTestExample.java` is a class with only a `main` method to test executing an SQL query with JDBC.
- If you haven't done so yet, change into the folder you have created.
- Enter:
  `export CLASSPATH=$CLASSPATH:.:mysql-connector-java-8.0.23.jar`
  If you log out you have to enter this again after the next login. To avoid having to enter it every time again, you can add the line to your `.bashrc`.
- Compile the Java classes (**`javac JDBCTestExample.java`** etc.)
- Execute an appropriate Java class with a `main` method, for instance `JDBCTestExample.java` (enter `java JDBCTestExample`).

Note:
JAR stands for Java Archive. It is a facility to bundle up multiple files into a single archive file. It works similar to the Unix utility `tar`.

2 Installation and Setup

(This chapter is relevant only if you want to install the environment on your own computer. The notes are not up to date with respect to versions, so check what is relevant today. For the lab exercises in the LIDA, this chapter is not necessary).

Installing the required environment with MySQL involves the following steps:

1. Install Java and JDBC on your machine
   To install both the Java platform and the JDBC API, follow the instructions for downloading the latest release of the JDK (Java Development Kit).
   When you download the JDK, you will get JDBC as well. You can find the latest release at the following URL: http://www.oracle.com/technetwork/java/javase/downloads/index.html

Install a JDBC driver on your machine
Your driver should include instructions for installing it. For JDBC drivers written for specific DBMSs, installation consists of just copying the driver onto your machine; there is no special configuration needed.
You can get the JDBC driver for MySQL and its documentation at the following URL:
http://dev.mysql.com/downloads/connector/j/

2. Install your own MySQL if needed
   If you do not already have a MySQL installed, you will need to follow the vendor's instructions for installation.

3. Setting up a database
   The following tutorial assumes that the database `koch_universitydb` already exists. First it is described how to open a connection with your MySQL, followed by some SQL code that can be sent to MySQL using JDBC. Finally, there is a description how to process the results that are returned.

3 Establishing a Connection to the Database

The first thing you need to do in a JDBC scenario is establish a connection with MySQL. This involves two steps: 1. loading the driver and 2. making the connection.

3.1 Loading a Driver

Loading the driver involves just one line of code, for instance:
`Class.forName("com.mysql.jdbc.Driver");`
or whatever the name of your driver is. In our example it is the above. Another example might be `org.gjt.mm.mysql.Driver`.

You do not need to create an instance of a driver and register it with the `DriverManager` because calling `Class.forName` will do that automatically. If you were to create your own instance, you would be creating an unnecessary duplicate, but it would do no harm.

3.2 Making the Connection to the Database

The second step in establishing a connection is to have the appropriate driver connect to the DBMS. The following Java statement illustrates the general idea:

```
Connection con = DriverManager.getConnection
                        (url, "myUsername", "myPassword");
```

If you are using the JDBC driver for MySQL, the JDBC URL will start with **`jdbc:mysql:`**. The rest of the URL is your data source name or database system. So, if you are accessing a MySQL database called "koch_universitydb", for example, your JDBC URL could be
`jdbc:mysql://193.196.143.168/koch_universitydb`

In place of "myUsername" you put the name you use to log in to MySQL; in place of "myPassword " you put your password for MySQL. So if you log in to your MySQL with the login name "koch" and the password "test, " the following two lines of code will establish a connection:

```
String url = "jdbc:mysql://193.196.143.168/koch_universitydb";
Connection con = DriverManager.getConnection
                (url, "koch", "test");
```

Note:
The IP address or name of the server only needs to be given in case the database is located on a different machine than the one you are working on, as we are doing here.

If you are logged into the machine on which the database is located, it would be sufficient to use `jdbc:mysql:koch_universitydb` as the URL.

If one of the drivers you loaded recognizes the JDBC URL supplied to the method `DriverManager.getConnection`, that driver will establish a connection to the DBMS specified in the JDBC URL. The `DriverManager` class manages all of the details of establishing the connection. Unless you are writing a driver, you will probably never use any of the methods in the interface `Driver`, and the only `DriverManager` method you really need to know is `DriverManager.getConnection`.
The connection returned by the method `DriverManager.getConnection` is an open connection you can use to create JDBC statements that pass your SQL statements to the MySQL. In the example here, `con` is an open connection.

4 Executing SQL Statements

1. Creating JDBC Statements
   SQL statements are sent to the DBMS by `Statement` objects.
   You create a `Statement` object and then execute it by using the appropriate `execute` method for the SQL statement you want to send.
   - For a **SELECT** statement, the method to use is **`executeQuery`**.
   - For statements with **CREATE** or **UPDATE** or **DELETE** or other statements that modify tables, the method to use is **`executeUpdate`**.
   To create a Statement object, you need an instance of an active connection. In the following example, we use the `Connection` object `con` to create the `Statement` object `stmt`:
   ```
   Statement stmt = con.createStatement();
   ```

2. Executing Statements
   After having created a statement, it can be executed by calling methods of the class Statement. The SQL statements are passed as String parameters to these methods. The method used most often for executing SQL statements is **`executeQuery`** which is needed for SELECT statements.
   Example:
   ```
   stmt.executeQuery("SELECT * FROM Student");
   ```

5 Retrieving Values from Result Sets

JDBC returns results in a `ResultSet` object, so we need to declare an instance of the class `ResultSet` to hold the results. The following code demonstrates declaring the `ResultSet` object `rs` and assigning the results of our earlier query to it:
```
ResultSet rs = stmt.executeQuery(" SELECT * FROM student ");
```

5.1 Iterating through the Result Tuples with the Method `next`

The variable `rs`, which is an instance of `ResultSet`, contains the rows of the table

Student. In order to access the values of matNr and sName, we will go to each row and retrieve the values according to their types.

The method next of the class ResultSet moves a cursor to the next row and makes that row (called the current row) the one upon which we can operate. 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.
The boolean return value of next informs whether there are any tuples left or not. next returns true if the cursor points to a real tuple after invocation of next. It returns false if the cursor has found no (additional) tuple.

5.2 Retrieving Column Values with the getXXX Methods

The current row contains a set of column values of various types. For each value, a method named getXXX of class ResultSet is used, where XXX is the SQL type of the column value.
getXXX delivers a return value with an appropriate Java type.
Examples:

- getInt can retrieve a value of SQL type INT and returns a Java int value. The first column (matNr) in each row of our result set rs is matNr, which holds a value of SQL type INT. Therefore getInt can be used.
- getString can retrieve a value of SQL type CHAR and returns a Java String object. The second column (sName) in each row of our example stores a value of SQL type CHAR. Therefore getString can be used.

getXXX takes a parameter to identify which column shall be addressed.
getXXX is overloaded so that the parameter can either be

- a String, giving the column name as it is in the database schema, or
- an integer, referring to the sequence of the column values in a tuple (1 for the first column, 2 for the second, etc.)

The following code accesses the values stored in the current row of rs and prints a line with the matNr followed by three spaces and the sName. Each time the method next is invoked, the next row becomes the current row, and the loop continues until there are no more rows in rs.

```
String query = "SELECT * FROM student";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
   int matrikel = rs.getInt("matNr");
   String name = rs.getString("sName");
   System.out.println("" + matrikel + " " + name);
} // end while
```

The output will look like this:

```
1234 Schmidt
2345 Schmidt
1000 Reinhard
4000 Wagner
```

JDBC offers two ways of identifying the column from which a getXXX method gets a value. One way is to give the column name as a String constant, as was done in the example above. The second way is to give the column index (number of the column), with 1 signifying the first column, 2, the second, and so on. Using the column number instead of the column name looks like this:

```
int matrikel = rs.getInt(1);
String name = rs.getString(2);
```

The first line of code gets the value in the first column of the current row of `rs` (column matNr) . The second line of code gets the value stored in the second column of the current row of `rs`.

Note:
- The column number refers to the column number in the result set, not in the original table.
- Using the column number is slightly more efficient, and there are some cases where the column number is required. In general, though, supplying the column name is essentially equivalent to supplying the column number. Also, it is somewhat more readable.

5.3 Methods to Retrieve JDBC Types

JDBC allows a lot of latitude as far as which `getXXX` methods you can use to retrieve the different SQL types.

Examples:
1. The method `getInt` can be used to retrieve any of the numeric or character types. The data it retrieves will be converted to an `int`; that is, if the SQL type is `VARCHAR`, JDBC will attempt to parse an integer out of the `VARCHAR`. The method `getInt` is recommended for retrieving only SQL `INTEGER` types, however, and it cannot be used for the SQL types `BINARY`, `VARBINARY`, `LONGVARBINARY`, `DATE`, `TIME`, or `TIMESTAMP`.

2. Although the method `getString` is recommended for retrieving the SQL types `CHAR` and `VARCHAR`, it is possible to retrieve any of the basic SQL types with it. (You cannot, however, retrieve the new SQL3 datatypes with it.) Getting all values with `getString` can be very useful, but it also has its limitations. For instance, if it is used to retrieve a numeric type, `getString` will convert the numeric value to a Java `String` object, and the value will have to be converted back to a numeric type before it can be operated on as a number. In cases where the value will be treated as a string anyway, there is no drawback. Further, if you want an application to retrieve values of any standard SQL type other than SQL3 types, use the `get-`

`String` method.

The following table shows which methods can legally be used to retrieve SQL types and, more importantly, which methods are recommended for retrieving the various SQL types. The SQL types are also called "JDBC types". Both terms refer to the generic SQL types defined in `java.sql.Types`.

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHARACTER	VARCHAR	LONGVARCHAR	BINARAY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	x	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	Y	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	Y	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	Y	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	Y	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	Y	Y	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	Y	Y	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	Y	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	Y	Y	x	x	x	x	x	x	x
getBytes														Y	Y	x			
getDate											x	x	x				Y		x
getTime											x	x	x					Y	x
getTimestamp											x	x	x				x	x	Y
getAsciiStream											x	x	Y	x	x	x			
getUnicode-Stream											x	x	Y	x	x	x			
getBinary-Stream														x	x	Y			

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHARACTER	VARCHAR	LONGVARCHAR	BINARAY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

A little "x" indicates that the `getXXX` method may legally be used to retrieve the given JDBC type.
A capital "Y" indicates that the `getXXX` method is recommended for retrieving the given JDBC type.

6 Summary

The descriptions in the previous sections explain the basics of JDBC, how to query a table and retrieve results. This can now be used in a program written in the Java programming language using the JDBC API. The examples in this document used only very simple queries, but as long as the driver and DBMS support them, one can send very complicated SQL queries using only the basic JDBC API.

Here is a simple complete example that you can run yourself.

```
import java.sql.*;
/**********************************************************
* JDBCTest is a sample class to show how to use JDBC to connect
* MySQL database.
*
* @version 08.10.2001
* @author Kexin Zhou
**********************************************************/
public class JDBCTest {

   public static void main(String args[]) {

      String dbDriver = "org.gjt.mm.mysql.Driver";
      String dbURL = "jdbc:mysql://localhost/koch_universityDB";
                /* Here you can provide the necessary URL, for instance
                   jdbc:mysql://193.196.143.168/koch_universitydb" */
      String user = "koch";
      String password = "test";
      Connection con = null;
      Statement stmt = null;
```

```
      try { // Loading the driver
         Class.forName(dbDriver);

         // Making the connection
         con = DriverManager.getConnection(dbURL, user, password);
         stmt = con.createStatement();

         } catch (ClassNotFoundException e) {
            System.out.println("Couldn't connect to MySQL Server");
            System.out.println(e.getMessage());
         } catch (SQLException e) {
            System.out.println("Couldn't access universityDB.");
            System.out.println("Maybe MySQL Server has been shut down!");
            System.out.println(e.getMessage());
         } // end try/catch Loading the driver

      String sqlStatement = "SELECT * FROM Student";
      ResultSet result = null;

      try {// Executing the query
         result = stmt.executeQuery(sqlStatement);
         // Retrieving and Printing out the results
         System.out.println("SQL statement: " + sqlStatement);
         System.out.println("matNr sName");
         while (result.next()) {
            int n = result.getInt("matNr");

            String s = result.getString("sName");
            System.out.println("" + n + " " + s);
         }// end while

         result.close();
         stmt.close();
         con.close();

      } catch (SQLException e) {
         System.out.println(e.getMessage());
      }// end try/catch Executing the query

      con.close(); //close the database connection

   } // end main
}//end JDBCTest
```

7 References

Maydene Fisher: JDBC(TM) Database Access