6. JDBC

JDBC – the Java Database Connectivity – is a standard vendor-independent API to connect a Java application to a database, provided a JDBC driver for this particular database is available.

JDBC appears to the application programmer as an API, i.e. a set of classes and methods that can be used to program an application without having to care about the specific properties of the underlying DBMS.

6.1 JDBC Drivers

JDBC drivers come in four different types (see the next figure).

JDBC Driver Architecture

From left to right the JDBC driver models are

- Type 1: The connection is established to a database equipped with an ODBC driver. ODBC stands for “Open Database Connectivity”, a standard to provide DB connections independently of programming language and operation system. JDBC/ODBC bridges are not very efficient and the complexity of the connection software can have negative impacts on the stability of the resulting system. Type 1 connections are only recommended when no other connection methods are feasible.
- Type 2: These connections mostly use “native” code in the driver to communicate directly with the database API. This means, native code would be part of the application. Type 2 drivers are faster than type 1 drivers.
- Type 3: These drivers use network protocols to communicate with a database middleware which translates messages into direct DBMS statements. This sounds like the most flexible solution because the middleware hides much of the internal workings of the database from the driver.
- Type 4: These drivers directly implement the DBMS API in Java – this generally makes them proprietary but very efficient. Most DBMS vendors supply type 4 drivers. For example, the MySQL JDBC driver com.mysql.jdbc.Driver is a Type 4 driver.

The basic blueprint for a JDBC application consists of four steps:

1. Find the correct driver for your database system.
2. Set up a connection to a particular database.
3. Execute a series of SQL statements.

4. Close the connection (and whatever else might be open.

6.1.1. Acquiring the driver

Technically, drivers are identified by a class that implements the java.sql.Driver interface, like the MySQL driver com.mysql.jdbc.Driver. The class is usually distributed in a .jar file, like mysql-connector-java-5.1.5-bin.jar**.**

Drivers are are managed by the the DriverManager class which creates a connection to a given database from an instance of the appropriate driver class.

To locate the driver and register an instance of it with the DriverManager class:

```
try {
        Class.forName("com.mysql.jdbc.Driver");
} catch (SQLException e) { ... }
```

For other databases, other driver class names have to be used, of course.[1]

5.1.2. Setting up a connection

The DriverManager class can now create a new connection using the driver just registered and a description of the database: the **database URL**. A DB URL is of the form

jdbc:subprotocol:description

For MySQL the subprotocol is – not surprisingly - "mysql" and the description is a string

//<host>:<port>/<database>?user=<user>&password=<password>

So, the full URL for the library database could look like

jdbc:mysql://localhost:3306/library?user=user1&password=user1pw

The connection itself is represented by an instance of a class implementing the Connection interface and is obtained by getting it from the DriverManager:

```
String dbURL =
```

1 If you became curious and looked up what the `Class`" class is and what the `forName` method does, this statement will probably be pretty unclear, to say the least. So you might be tempted to take this statement as a kind of incantation and just write it down or you will have to figure out what's really going on there.

```
        “jdbc:mysql://localhost:3306/library?user=user1&password=user1pw”;
try {
        java.sql.Connection conn = DriverManager.getConnection(dbURL);
        ...
} catch (SQLException e) {...}
```

Ok, it's getting a bit tricky now: There are quite a few Connections around, like java.sql.Connection or the deceptively similar com.mysql.jdbc.Connection. Importing or using the wrong Connection-typed variable might result in strange error messages and the need to cast one connection into another. Make sure you got the one from the java.sql package. And of course, you then would rewrite this as

```
import java.sql.Connection;

...

String dbURL =
        “jdbc:mysql://localhost:3306/library?user=user1&password=user1pw”;
try {
        Connection conn = DriverManager.getConnection(dbURL);
        ...
```

5.2 Executing SQL statements

Once the connection is set up a Statement instance is used to define and execute SQL statements:

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

The same precaution as above applies here: use java.sql.Statement, not something like com.mysql.jdbc.Statement as an import or name qualifier.

Depending on what SQL statement it represents, there are two main execution methods in the Statement interface:

- executeQuery(String sqlStatement) returning a ResultSet object; typically the sqlStatement is a SELECT,
- executeUpdate(String sqlStatement) returning an int; the sqlStatement arguments could be an INSERT, UPDATE or DELETE statement or a DDL statement.

Here's an example for an executeQuery:

```
ResultSet rs = stmt.executeQuery("SELECT * from Student");
```

ResultSets are used to give access to the result of the query:

> *"A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set."*

*[*http://java.sun.com/j2se/1.4.2/docs/api/java/sql/ResultSet.html]

Once the cursor is moved to a certain row, the row contents can be accessed by a variety of "get..." methods. Columns can be accessed by column number (starting with 1) or attribute name. So, just running through the result set could look like this:

```
while (rs.next()) {
        System.out.print(rs.getInt("StudentID"));          // int with name
        System.out.print(" " + rs.getString("Name"));      // String with name
        System.out.print(" " + rs.getString(3));           // String with number
        System.out.print(" " + rs.getInt(4));              // int with number
        System.out.println(" " + rs.getString("Major"));   // String with name
}
```

would produce the output

```
1 John Doe 4711 25 Computer Science
2 Suzy Creamcheese 388-1200 54 null
3 Jane Doe 555-19224 27 History
4 Gerald Bostock 123123-99 45 null
```

Producing a list of books and who borrowed them requires some SQL but not really a lot:

```
rs = stmt.executeQuery("SELECT CatalogNumber, Title, UwfID, Name
                                    FROM Book, Student
                                    WHERE Book.BorrowerID = Student.StudentID");
while (rs.next()) {
        System.out.print(rs.getString(1));
        System.out.print(" " + rs.getString(2));
        System.out.print(" " + rs.getString(3));
        System.out.println(" " + rs.getString(4));
}
```

This produces

```
TK4412 War and Peace 555-19224 Jane Doe
```

6.1.4 Updates and Transactions

To study updates, let's look at a more involved example: Write a method that looks up a student by name, looks up a book by catalog number and places the StudentID of this student as a foreign key into the BorrowerID column of the book – i. e. we make the student borrow the book.

For this we provide

- Exception classes to indicate a student and/or a book not begin found
- Transaction protection for the operation

To start a transaction, we simply override the MySQL autocommit for the connection, in case of a problem, we roll back, else we commit at the end:

```
private static void borrowBook(Connection conn,
                                    String studentName, String catNum)
        throws SQLException,
```

```
            StudentNotFoundException,
            BookNotFoundException {

        conn.setAutoCommit(false); // BEGIN transaction


        Statement stmt = conn.createStatement();

        // Grab the ID (primary key) of the student with the given name
        ResultSet rs = stmt.executeQuery(
                "SELECT StudentID from Student " +
                        "WHERE Name = \"" + studentName + "\"");

        // Is there such a student?
        if (!rs.next()) {                       // Nope! Do something sensible...
                conn.rollback();                        // like rolling back and throwing
                conn.setAutoCommit(true);       // an exception
                throw new StudentNotFoundException("Student name " +
                                studentName + " not found");
        }


        // Update the foreign key in the Book relation
        int rowsAffected = stmt.executeUpdate(
                "UPDATE Book" +
                " SET BorrowerId = " + rs.getString(1) +
                " WHERE CatalogNumber = \"" + catNum + "\"");

        // Did it work?
        if (rowsAffected == 0) {  // Nope! Not our day, it seems...
                conn.rollback();
                conn.setAutoCommit(true);
                throw new BookNotFoundException("Book catalogNumber " +
                                catNum + " not found");
        }

        // Gotcha!
        conn.commit();
}
```

6.1.5 Prepared Statements

Prepared statements play an important practical role in JDBC programming. They describe a partially specified SQL statement – one in which parts have been replaced by placeholders or parameters - which is later completed by specifying parameter values.

The rationale behind prepared statements is to let the SQL compiler and the query optimizer process the basic structure of the SQL statement first and the let the optimized statement be executed multiple times. In this example, it's not making a whole lot of sense, since each statement is executed exactly once, but getting used to prepared statements is more important. As a rule of thumb, prepared statements are generally preferable to ad-hoc queries.

The queries in the above program would now be written as follows: the old version

```
ResultSet rs = stmt.executeQuery(
                "SELECT StudentID from Student ” +
                        “WHERE Name = \"" + studentName + "\"");
```

becomes

```
PreparedStatement pStmt =
              conn.prepareStatement(
                        "SELECT StudentId from Student WHERE Name = ?");
pStmt.setString(1,studentName);
ResultSet rs = pStmt.executeQuery();
```

As can be seen the parameters are written as question marks and filled in by numbers. A nice side effect is that these statements are somewhat easier to read and write.

The effect of multiple parameters can be seen by looking at the UPDATE. The old version

```
int rowsAffected = stmt.executeUpdate(
        "UPDATE Book" +
        " SET BorrowerId = " + rs.getString(1) +
        " WHERE CatalogNumber = \"" + catNum + "\"");
```

becomes

```
pStmt = conn.prepareStatement(
                "UPDATE Book SET BorrowerId = ? WHERE CatalogNumber = ?");
pStmt.setString(1,rs.getString(1));
pStmt.setString(2,catNum);
int rowsAffected = pStmt.executeUpdate();
```