Teaching Assistant **Andrei Vasilateanu**

Accessing databases in Java using JDBC

Introduction

JDBC is an API for Java that allows working with relational databases. JDBC offers the possibility to use SQL statements for DDL and DML statements. This API can be regarded as an abstraction layer between the application and the DBMS, programmers being able to work with the database independent of the underlying DBMS. Thus the code does not suffer modifications if the DBMS is changed. This independence is achieved by the fact that JDBC uses drivers, which translate JDBC API method calls to specific DBMS API. When we change the DBMS it is enough to change the driver, which can be done even at runtime.

(Note: Database notions are assumed to be known by the students, this lesson is focused only on using JDBC).

In addition to executing SQL statements, JDBC allows us to invoke stored procedures which are specific to the used DBMS.

Requirements

In this exercise we are using an the relational database provided with Java SDK, JavaDB (available for download from http://developers.sun.com/javadb/) By default it is installed in /Program Files/Sun/JavaDB.

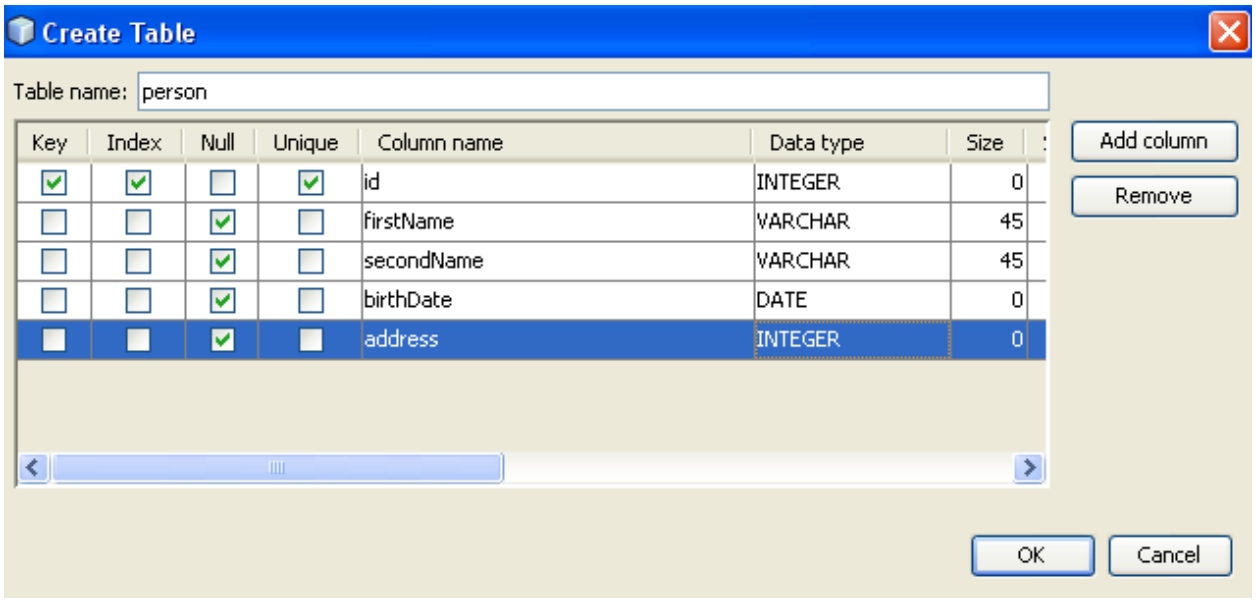
The IDE used is Netbeans (http://netbeans.org/)

Creating the database

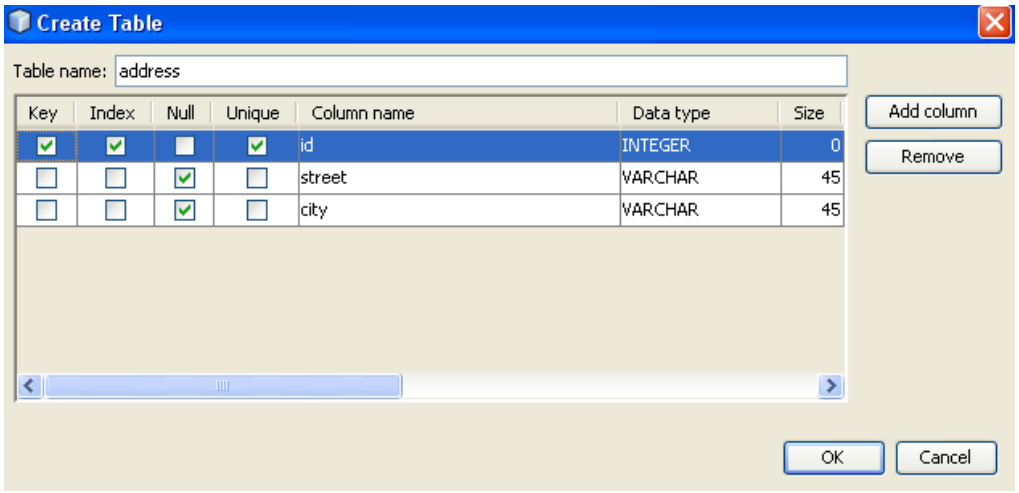
Start Netbeans

Go to Services->Databases. Start JavaDB. Create a database called “sdm” with username “a” and password “a”. Connect to the new database, expand it and select “APP” as the default schema (for Netbeans 6.7). Expand “APP” , right-clicl “Tables” and create a new table “person”

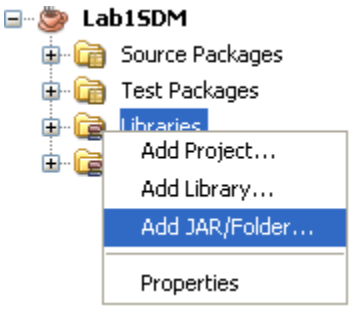
Create the person table:



Create the address table:

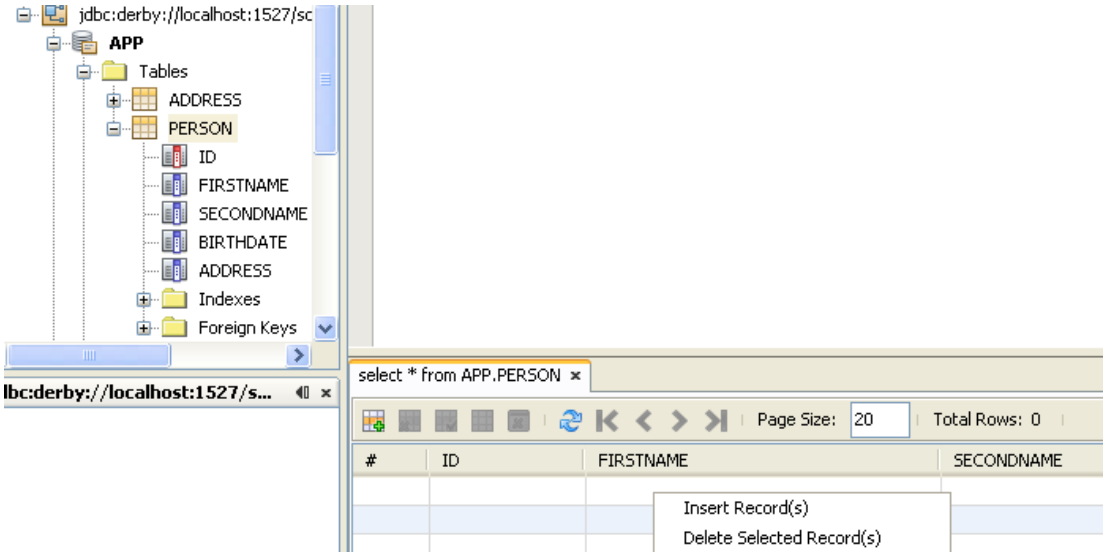


Create a new Java Project. In Libraries import the driver for the database “derbyclient.jar”



(should be in JavaDb/lib folder)

Add data to the Database using InsertRecord



Accessing a database using JDBC

Basic steps

We go through the following steps for accessing a database:

1. Making the connection

A connection is identified by an URL with the syntax:

jdbc:<protocol>:<nameOfDatabase>

In our case the URL has the form:

```
jdbc:derby://server:port/nameOfDatabase
```

We also need a username and a password for connecting to the database (In our case the default is username = a, and the password is a). This three arguments (nameOfDatabase, user, password) should not be hardcoded in the application but instead saved in an external file (for example using java.util.Properties) so you can have a reusable class for connecting to the database.

So let`s save the url in a string:

```
String url="jdbc:derby://localhost:1527/sdm";
```

(note that the port number is 1527 for JavaDB)

and then connect to the database:

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

2. Execute an SQL statement

First we create a Statement from the Connection:

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

then we execute this statement. In case of a SELECT statement we use executeQuery() which returns a ResultSet and in case of an UPDATE, DELETE, INSERT, CREATE TABLE, DROP TABLE we use executeUpdate() which returns the number of affected rows. (except for create and drop where it returns 0).

3. Retrieve values from the ResultSet

JDBC returns results in a ResultSet object:

```
String sql="SELECT * FROM app.person p";
```

```
ResultSet rs=instr.executeQuery(sql);

while(rs.next()){

System.out.println(rs.getString(2));

}
```

rs.next() points the cursor of the resultset to the next position (initially it is before the first record). The values are extracted using getx() methods where x is the data type and the argument is the position of the column in the table. We could have also used the name of the column :

```
System.out.println(rs.getString("firstName"));
```

4. Close the connection

ResultSet, Statement must be closed first:

```
rs.close();

instr.close();
```

and then the Connection:

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

The complete program :

```
import java.sql.*;

public class Main {

   public static void main(String[] args) throws ClassNotFoundException, SQLException {

      String url = "jdbc:derby://localhost:1527/sdm";
      Connection con = DriverManager.getConnection(url, "a", "a");
      Statement instr = con.createStatement();
      String sql = "SELECT * FROM app.person p";
      ResultSet rs = instr.executeQuery(sql);
      while (rs.next()) {
         System.out.println(rs.getString(2));
      }
      rs.close();
          instr.close();
      con.close();
   }
}
```

Using JOIN

Suppose you want to use more tables to perform a certain operation. In this case you join the tables using a value they have in common. (usually a foreign key). In our case we join person and address using person.address and address.id fields:

```
String sql="SELECT p.firstName FROM app.person p, app.address a where
p.address = a.id and a.city = 'Bucuresti'";
```

Using transactions

A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed. By default the envriroment is non-transactional so each statement is executed immediately. To modify this we use :

```
con.setAutoCommit(false);
```

In this case the statements will be executed only when we commit them:

```
con.commit();
```

To be sure that if one statement fails in a transaction, the database is returned to its previous state we use rollback() method. This method should be executed in the catch block:

```
catch (SQLException e) {
                    e.printStackTrace();
                    con.rollback();
            }
```

SQL quick notes

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_option] ...

create_definition:
    col_name column_definition
  | [CONSTRAINT [symbol]] PRIMARY KEY [index_type] (index_col_name,...)
        [index_type]
  | {INDEX|KEY} [index_name] [index_type] (index_col_name,...)
        [index_type]
  | [CONSTRAINT [symbol]] UNIQUE [INDEX|KEY]
        [index_name] [index_type] (index_col_name,...)
        [index_type]
```

```
  | {FULLTEXT|SPATIAL} [INDEX|KEY] [index_name] (index_col_name,...)
        [index_type]
  | [CONSTRAINT [symbol]] FOREIGN KEY
        [index_name] (index_col_name,...) reference_definition
  | CHECK (expr)

column_definition:
    data_type [NOT NULL | NULL] [DEFAULT default_value]
      [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]
      [COMMENT 'string'] [reference_definition]


DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]

INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name [(col_name,...)]
    {VALUES | VALUE} ({expr | DEFAULT},...),(...),...
    [ ON DUPLICATE KEY UPDATE
      col_name=expr
        [, col_name=expr] ... ]




SELECT
    [ALL | DISTINCT | DISTINCTROW ]
      [HIGH_PRIORITY]
      [STRAIGHT_JOIN]
      [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
      [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [FROM table_references
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [PROCEDURE procedure_name(argument_list)]
    [INTO OUTFILE 'file_name' export_options
      | INTO DUMPFILE 'file_name'
      | INTO var_name [, var_name]]
    [FOR UPDATE | LOCK IN SHARE MODE]]

UPDATE [LOW_PRIORITY] [IGNORE] table_reference
    SET col_name1={expr1|DEFAULT} [, col_name2={expr2|DEFAULT}] ...
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```