CS1003 Week 7 Practical: JDBC

This practical is worth **20%** of the overall coursework mark. It is due at **9pm** on **Thursday 14th March (week 7)**. As for every practical, you should arrive in the lab having prepared in advance, by studying this specification and reviewing relevant course material. Unless otherwise stated, please use **Java** for basic requirements and extension activities on this module.

Skills and Competencies

- developing robust software to manipulate data stored in a database
- choosing an appropriate way to store and manipulate data in your program and in the database
- choosing appropriate classes and methods from the JDBC API
- identifying and dealing with possible error conditions
- testing and debugging
- writing clear, tidy, consistent and understandable code

Setting Up

As in previous practicals, we are using the automated checker `stacscheck`. For this practical, your program must have a `main` method in a class called `W07Practical` which is in a file `W07Practical.java` that is located inside your `src` directory in your `W07-Practical` directory.

If you can't remember how to create a suitable project setup, have a look at "Setting Up" in:

https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W03/W03-Practical-File-Processing.pdf

You will also need some other files from StudRes at

https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W07/

To start, you should download the `sqlite-jdbc.jar` file to your `W07-Practical/src` directory.

Requirements

The practical involves processing the same data as in the Week 3 Practical. As before, various versions of the data file, of different lengths, can be downloaded from:

- https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W03/data/clean_small.csv
- https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W03/data/clean_large.csv

You are strongly advised to do all your initial development and debugging using the *small* version.

Write a Java program using JDBC, to read a given *data file*, store some of the data columns in a SQLite database and perform certain queries over the data in the database. Your program should:

- take *four* command-line arguments (by making use of the `args` parameter to your `main` method in your `W07Practical` class).
  - `args[0]` should represent the path of the *filename* of the *sqlite* database file.
  - `args[1]` should specify the action taken by your program and can be one of:
    `{ create, query1, query2, query3, query4, query5 }`
  - `args[2]` is only needed for the `create` , `query2`, and `query3` actions
    - for `create` it should represent the path to the CSV data file from which your program will read the data and insert it into the sqlite database
    - for `query2` and `query3` it should represent the *Minimum Rating* to use.

- Connect to the SQLite database specified by the *filename* in `args[0]`
    - for example, to establish a connection to a sqlite database `"test.db"` in the current directory on the Linux lab machines, you can use the following
      ```
      Connection connection = DriverManager.getConnection("jdbc:sqlite:test.db");
      ```
    - study the JDBC lectures and examples on StudRes to see what you can do once you have a connection set up.
- For the `create` action:
    - check whether a table for the data already exists in the database, and if so, delete it
    - create a table with suitable columns and types – we suggest you should use an `integer unsigned` type for `number`, suitable SQL `varchar` types for `name`, `city`, `cuisine style`, and the SQL `double` type for `rating`.
    - read the `number`, `name`, `city`, `cuisine style`, and `rating` data from the input file specified in `args[2]` (e.g. `clean_small.csv`) and insert it into the table – **Note**, missing values for *cuisine style* and *rating*, and *ratings* **< 0** should be entered as (set to the special value) `null` in the database (not the string "null"), signifying that the value is in fact unknown. You will also have to make sure you are dealing with null values properly when querying the database.
    - print out "OK" if the action was successful.
    - **Note**: In order to make insertion into the table more efficient (especially for the large data file) you should probably turn off auto commit on your open *sqlite* database `Connection` object prior to executing your insertion statements and then manually commit afterwards by
      ```
      connection.setAutoCommit(false);
      // execute all insertion statements here
      connection.commit();
      ```
- For the query actions, perform queries on the table to:
    - `query1:` list the *city*, *name*, *rating* and *cuisine_style* for the best rated restaurants in `Amsterdam` and `Edinburgh` (with rating 5) which serve `European` cuisine.
    - `query2:` print out the total number of restaurants with a rating greater or equal to the minimum rating given by `args[2]`.
    - `query3:` print out a table, containing for each *city*, the *city* and number of restaurants with a rating greater or equal to the minimum rating given by `args[2]`.
    - `query4:` print out a table, containing for each *city*, the *city* and average *rating* of restaurants in that city.
    - `query5:` for each *city*, list the *city*, restaurant *name* and *rating* for the lowest rated restaurants in that city – you cannot assume that different cities have the same minimum rating value.
    - As for the week 3 practical, all numbers should be printed with up to 1 decimal place using e.g. the `DecimalFormat` class, no rounding is required.
- After successfully completing any one of the actions above, your program should exit – it is envisaged that you would run your program initially to `create` the database from the given CSV file, and then subsequently run it again using `query` actions to query the database that was created during `create`.

For the `query` parts of your attempt, you should use JDBC and queries to get the results you need from the database, i.e. your program should not need to access the CSV data file again. You can use example code from

StudRes as a starting point. You should clearly identify using in-line comments in your source code and in your report, which files or code fragments you have modified from that which was given out in class or which you have sourced from elsewhere.

Compiling and Running your program

In order to be compiled by the stacscheck auto checker, it must be possible to compile your program from the `src` directory (which is assumed to also contain the `sqlite-jdbc.jar`) using the command

```
javac -cp sqlite-jdbc.jar:. *.java
```

Below are some examples showing how your program should be run and the expected output for some different command-line arguments indicating the functionality that you should provide. These commands also assume your current directory is the `src` directory which also contains the small data file and the `sqlite-jdbc.jar` and that commands are run in the order shown below.

```
java W07Practical
Usage: java -cp sqlite-jdbc.jar:. W07Practical <db_file> <action> [input_file | minimum_rating]

java -cp sqlite-jdbc.jar:. W07Practical test.db create clean_small.csv
OK

java -cp sqlite-jdbc.jar:. W07Practical test.db query1
city, name, rating, cuisine_style
Amsterdam, Martine of Martine's Table, 5, ['French'; 'Dutch'; 'European']
Amsterdam, Vinkeles, 5, ['French'; 'European'; 'International'; 'Contemporary'; 'Vegetarian Friendly'; 'Vegan
Options'; 'Gluten Free Options']

java -cp sqlite-jdbc.jar:. W07Practical test.db query2 3
Total number of restaurants with rating above (or equal to) 3
10

java -cp sqlite-jdbc.jar:. W07Practical test.db query3 3
city, number of restaurants with rating above (or equal to) 3
Amsterdam, 5
Athens, 5

java -cp sqlite-jdbc.jar:. W07Practical test.db query4
city, average rating
Amsterdam, 4.7
Athens, 4.3

java -cp sqlite-jdbc.jar:. W07Practical test.db query5
city, name, rating
Amsterdam, De Silveren Spiegel, 4.5
Amsterdam, La Rive, 4.5
Amsterdam, Librije's Zusje Amsterdam, 4.5
Athens, Hermion, 3.5
```

The first invocation above does not pass any command-line arguments to the program and results in the required usage message being displayed in the terminal window. The second invocation does pass the expected command-line arguments to the program for `create` and as a result produces some console output indicating that the action was successful. The other invocations above, run the various queries for the small data file and show the expected output. To compare your program's output against what is expected for `clean_large.csv`, you can run the *stacscheck* auto checker as shown below.

The program should produce the output exactly as shown above and such output will be expected by the auto checker. Your program should deal gracefully with possible errors such as the input file being unavailable, or the file containing data in an unexpected format.

Running the Automated Checker

As usual, please use the automated checker to help test your attempt. Please make sure you have a `src` directory containing

- all your source code including a `W07Practical` class containing your `main` method in `W07Practical.java`
- the `sqlite-jdbc.jar` file

If your `src` directory does not contain these the auto checker will fail.

The checker is invoked from the command line on the Linux Lab Machines in your `W07-Practical` folder:

```
stacscheck /cs/studres/CS1003/Practicals/W07/Tests
```

As in previous practicals, if some of the tests are failing, examine the output and try to identify why your program is not compiling, running, or producing the output as expected. If nothing is working and you don't know why, please don't suffer in silence, ask one of the demonstrators in the lab.

Testing

You are encouraged to create and document your own tests to test how your program deals gracefully with possible errors such as the input file being unavailable, or the file containing data in an unexpected format. To prove that you have tested the above scenarios, paste terminal output from each of your tests into your report in the Testing section. Be clear what each test demonstrates.

Deliverables

Hand in via MMS to the *Week 7 Practical* slot, a `.zip` file containing your entire `W07-Practical` directory which should contain:

- All your Java source files in the `src` directory as specified above.
- You may submit created sqlite database files if you wish, but our auto checker tests will run your program on a different database, not the one you submit to MMS.
- A **PDF** report containing the usual sections for *Overview*, *Design & Implementation*, *Testing*, *Examples* (example program runs), *Evaluation* (against requirements), *Conclusions* (your achievements, difficulties and solutions, and what you would have done given more time). Your report should also contain an accurate summary stating which files or code fragments you have written and any code you have modified from that which was given out in class or which you have sourced from elsewhere.
- Please include in your report, output from running the auto checker on your program using your own database.

Extensions

If you wish to experiment further, you could try any or all of the following:

- Alter your program to accept additional actions to present additional interesting output or statistics by using many interesting queries and explain and discuss your results in the report.
- Investigate and make use of aggregate (*group by*) functions and sub-queries in SQL such that the results of your queries are always computed on the database and your program merely needs to execute the query and print out the result for each. You might wish to look at the documentation at:

  http://www.sqlitetutorial.net/sqlite-group-by/

- Investigate the use of *views* in the database.

http://www.sqlitetutorial.net/sqlite-create-view/

Your program could create a suitable view for each of your queries in the database and then use these views to display the query results. This would permit you to re-run the same queries created by your program by performing a simply *select* query on the views from within the `sqlite3` command-line tool.

Don't forget to document your extension work in your report. You are also free to implement your own extensions, but clearly state these in your report. If you use any additional libraries to implement your extensions, ensure you include these in your submission, with clear instructions to the marker on how to resolve any dependencies and run your program.

Marking

Your submission will be marked using the standard mark descriptors in the School Student Handbook:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

However, more specifically, for this practical:

- 1-6: Very little evidence of work, software does not compile or run, or crashes before doing any useful work. You should seek help from your tutor.
- 7-10: A decent attempt which gets part of the way toward a solution, but has serious problems such as not compiling (at the low end), or lacking in robustness and maybe sometimes crashing during execution (at the high end).
- 11-13: A solution which is mostly correct and goes some way towards fulfilling basic requirements, but has issues such as not always printing out the correct values.
- 14-15: A correct solution accompanied by a good report, but which can be improved in terms of code quality, for example: poor method/OO structure, lack of comments, or lack of reuse.
- 16-17: A very good, correct solution to the main problem containing clear and well-structured code achieving all required basic functionality, demonstrating very good design and code quality, good method and class decomposition, elegant implementation of methods with reuse where appropriate, and accompanied by a well-written report.
- 18-19: As above, but implementing at least one or more extensions, accompanied by an excellent report. However, as mentioned before, it is key that you focus on getting a very good solution to the basic requirements prior to touching any of the extension activities. A weak solution with an attempt at extensions is still a weak solution.
- 20: As above, but with multiple extensions, outstanding code decomposition and design, and accompanied by an exceptional report.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:

http://www.st-andrews.ac.uk/students/rules/academicpractice/

Hints

Here is one possible sequence for developing your solution.

1. You should first design a relational schema to represent the data in the database. Such a schema should detail your table name, column names and types.
2. Use the *sqlite3* command line interface to design and test an SQL command to create the table with the appropriate structure.
3. Examine the Week 6 examples on StudRes. These example programs show how you can connect to a database, create a table, add data and run a simple query.
4. Write a Java program that creates a JDBC connection and executes the SQL command. Check via the *sqlite3* command line interface that the table has been created successfully.
5. Repeat step 2 to design and test an SQL command to insert a new row into the table.
6. Refine your program to include the above command, with some dummy data values. Test and debug as necessary.
7. Add code to open an input file (e.g. clean_small.csv) and iterate through it line by line.
8. You could write a method `String makeRowInsertStatement(String[] elements)` that takes the required elements from one line of the data file, and creates an SQL command to insert these elements into the table.
9. You might call this method within a loop that iterates through the data file, so that an SQL insertion is executed for each row in the file.
10. Test your program on the small data file, and check that the table has been correctly populated using the *sqlite3* command line interface.
11. Write a method `executeQueryX` which can execute an SQL query and pass the resulting `ResultSet` object to a `void printResultSet(ResultSet rs)` method which can print the results to the console. For the latter method, you will need two nested loops: an outer loop that iterates through the rows of the query result, and an inner loop that iterates through each column value for that row.
12. Repeat the previous step for the other queries and think about the queries you want to execute and how to find the information required using a combination of Java and SQL.