Lab: Java Database Connectivity (JDBC)

The goal of the lab is to write a utility class named `DataAccess` that enables an application to transparently access its data, using high-level Java methods. The `DataAccess` class is meant to hide from the application how its data is actually stored (here, in a relational database) as well as all the complex machinery required to access it (here, JDBC library and SQL statements).

The role of the application is played by the `main` method of some other class, e.g. `Test`. When launched, the application first creates an object of the `DataAccess` class. It then uses this object to read or write data to the database using the high-level methods developed in the exercises below.

We take as an example the Company database of lab 2. The `DataAccess` class provides methods for retrieving the employee list, update their salary and so forth.

Please refer to the Java API documentation for more information about the JDBC classes we'll be using: http://docs.oracle.com/javase/8/docs/api/. They all belong to the ***java.sql*** package.

Preamble

You may use the IDE of your choice to develop the application, e.g. NetBeans, Eclipse, etc. You must first create a plain Java project. Then you need to set up your execution environment by performing the following steps.

Installing the MySQL driver

MySQL's JDBC driver is named `com.mysql.jdbc.Driver`. It is shipped in a jar file named `mysql-connector-java-x.y.z.jar`, which you can download from Campus. You must add this jar file to the libraries of your project: this will include the jar file in the `classpath` of the project.

Setting the name of the driver

Next, you have to tell the JDBC runtime which driver you want to use. This is done through the `jdbc.drivers` system property, which you must set to the name of the driver.

You can do so by executing the following instruction in your code:

```
System.setProperty("jdbc.drivers", "com.mysql.jdbc.Driver");
```

Better yet, you can set the `jdbc.drivers` property as a VM option when launching the application. If you launch it from the command line, you must use the following syntax:

```
java –Djdbc.drivers=com.mysql.jdbc.Driver <class name>
```

If you launch the application from your IDE, you must set the system property in the "VM options" section of the "Run" configuration of your project, as follows:

```
-Djdbc.drivers=com.mysql.jdbc.Driver
```

Setting the database's URL

The JDBC URL format for the MySQL's driver is as follows:

```
jdbc:mysql://[host][:port]/[database]
```

Use the values defined in the "How to connect to your database" tutorial for the `host`, `port` and `database` parts of the URL.

Miscellaneous

Development tip: to avoid writing `try/catch` clauses everywhere in your code, add the `throws SQLException` clause to all the methods / constructors that you develop, including `main`.

Exercise 1

Write the constructor of the class. The constructor takes the database's URL, the user's login and password as parameters and establishes a connection to the database. The connection is stored in a (private) instance field, since all the other methods of the class use the connection.

The parameters of the constructor are taken from the `String[] args` parameter of the `main` method. When using NetBeans, you can set these arguments in the "Arguments" section of the "Run" configuration of your project: in the corresponding field, list all the arguments on a single line, separated with space characters.

Next, write the `close()` method, which closes the connection to the database and releases any resource associated with it. The application must call this method when it is done using the database.

Exercise 2

Write the method `List<EmployeeInfo> getEmployees()` that returns the number, name and salary of all the employee in the EMP table. Note: the class `EmployeeInfo` is already defined in the `model` package.

Exercise 3

Write the method `boolean raiseSalary(String job, float amount)` that raises the salary of the employees with the specified job by the specified amount. Next, perform an SQL injection attack that raises the salary of *all* employees.

Exercise 4

Write a second version of the `getEmployees` and `raiseSalary`, named `getEmployeesPS` and `raiseSalaryPS`, that uses *prepared statements* instead of statements. Why prepared statements are more efficient? How exactly should you write your code to take advantage of them? Check that the SQL injection attack of the previous exercise is no longer possible.

Exercise 5

Write the method `List<DepartmentInfo> getDepartments(Integer id, String name, String location)` that retrieves the departments matching the specified criteria. A criterion may be omitted by specifying the Java `null` value. Here are some examples of use:

- `getDepartments(null, null, null)` retrieves all departments
- `getDepartments(null, null, "NEW-YORK")` retrieves all the departments located in New-York
- `getDepartments(null, "RESEARCH", "DALLAS")` retrieves all the departments named Research **and** located in Dallas

Develop two implementations of the method: one with statements and the other one with prepared statements.

Exercise 6

Write the method `List<String> executeQuery(String query)` that executes the specified query (i.e. `select` statement) on the database. The methods returns a list of strings, each representing a line of the result displayed by some interactive tool – mysql, MySQLWorkbench, etc. – executing the same query: the first string lists the attributes of the result; each subsequent string represents one tuple of the result.

Write the method `List<String> executeStatement(String statement)` that executes any statement (e.g. `select`, `insert`, `update`, etc.) on the database. If the statement is a query, the method returns the same value as `executeQuery()`; if the statement is an update, the method returns a singleton list that contains the number of tuples that were updated.

Can you use prepared statements to implement the above methods? Explain.