Assignment 2

SQL Programming

Due date: 20.12.20

Submission is in pairs.
Please use hw2's piazza forum for any question you may have.

1. Introduction

You are about to take a lead part in the development of the **"OnlineTest"** database, a website that holds information about tests, tests' locations, students and supervisors.

In **OnlineTest**, users with admin privileges (you), can add a test located in a specific zoom room, add a student, confirm the supervisors for the test and so on.

OnlineTest is a smart service that gives you statistics about tests and recommends courses based on people's rating for the matching tests.

Your mission is to design the database and implement the data access layer of the system. Typically, the data access layer facilitates the interaction of other components of the system with the database by providing a simplified API that carries out a predefined desired set of operations. A function in the API may receive business objects as Input arguments. These are regular Java classes that hold special semantic meaning in the context of the application (typically, all other system components are familiar with them). The ZIP file that accompanies this document contains the set of business objects to be considered in the assignment, as well as the full (unimplemented) API. Your job is to implement these functions so that they fulfill their purpose as described below.

Please note:

1. The database design is your responsibility. You may create and modify it as you see fit. You will be graded for your database design, so bad and inefficient design will suffer from points reduction.

2. Every calculation involving the data, like filtering and sorting, must be done by querying the database. You are prohibited from performing any calculations on the data using Java. Additionally, when writing your queries, you may only use the material learned in class. Furthermore, you cannot define your own classes, your code must be contained in the functions given except for the case of defining basic functions to avoid code duplication.

3. It is recommended to go over the relevant Java files and understand their usage.

4. All provided business classes are implemented with a default constructor and getter\setter to each field.

5. You may not use more than **one** SQL query in each function implementation, views in use at two or more functions are not included.

2. Business Objects

In this section we describe the business objects to be considered in the assignment.

Test

Attributes:

Description	**Type**	**Comments**
Test's course number (ID)	Integer	The course number.
Semester	Integer	The semester of the test (1 – Winter, 2 - Spring, 3 - Summer).
Time	Integer	Test's time (1 – 09:00, 2 – 13:00, 3 – 17:00)
Room	Integer	The room that the test is taking place in (all tests are in zoom).
Day	Integer	The day of the test (1,2 ,… ,31)
Credit Points	Integer	Number of points given for passing the course.

Constraints:

1. (ID, Semester) is unique across all tests.
2. IDs, Credit Points and Room are positive (>0) integers.
3. All attributes are not optional (not null).

Notes:

1. In the class Test you will find the static function badTest() that returns an invalid test.

Student

Attributes:

Description	**Type**	**Comments**
Student ID	Integer	The ID of the student.
Student Name	String	The name of the student
Faculty	String	Student's faculty.
Credit Points	Integer	The amount of credit points a student has.

Constraints:

1. IDs are unique across all students.
2. IDs are positive (>0) integers and Credit Points is not negative.
3. All attributes are not optional (not null).

Notes:

1. In the class Student you will find the static function badStudent() that returns an invalid Student.

Supervisor

Attributes:

Description	**Type**	**Comments**
Supervisor ID	Integer	The ID of the Supervisor.
Supervisor Name	String	The name of the Supervisor
Salary	Integer	The salary of the Supervisor per test.

Constraints:

1. IDs are unique across all supervisors.
2. IDs are positive (>0) integers.
3. Salary is not negative.
4. All attributes are not optional (not null).

Notes:

1. In the class Supervisor you will find the static function badSupervisor() that returns an invalid Supervisor.

Please pay attention, you are given with an initial state of the database – table that specifies how many credit points a student needs to graduate in respect to his/her faculty. Make sure to use your tests as in the example tests.

3. API

3.1 Return Type

For the return value of the API functions, we have defined the following enum type:
ReturnValue (enum):

- OK
- NOT_EXISTS
- ALREADY_EXISTS
- ERROR
- BAD_PARAMS

In case of conflicting return values, return the one that appears first.

3.2 CRUD API

This part handles the CRUD - Create, Read, Update and Delete operations of the business objects in the database. Implementing this part correctly will lead to easier implementations of the more advanced APIs.
You can assume the arguments to the function will not be NULL, the inner attributes of the argument might consist of NULLs.

ReturnValue addTest(Test test)

Adds a **test** to the database.
Input: Test to be added.
Output: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters.
* ALREADY_EXISTS if a test with the same ID and Semester already exists
* ERROR in case of a database error

Test getTestProfile(Integer testID, Integer Semester)

Returns the test profile of **testID**
on **Semester**.

Input: Test ID and semester.
Output: The test profile (a test object) in case the test exists. BadTest() otherwise.

ReturnValue deleteTest (Integer testID, Integer Semester)

Deletes a test from the database.
Deleting a test will delete it from everywhere as if it never existed.
Input: Test ID and semester to be deleted.
Output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if test does not exist
* ERROR in case of a database error

ReturnValue addStudent (Student student)

Adds a student to the database.
Input: Student to be added.
Output: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters
* ALREADY_EXISTS if a student with the same ID already exists
* ERROR in case of a database error

Student getStudentProfile (Integer studentID)

Returns the student with studentID as its id.

Input: Student id.
Output: The student with studentID if exists. BadStudent() otherwise.

ReturnValue deleteStudent (Integer studentID)

Deletes a student from the database.
Deleting a student will delete it from everywhere as if he/she never existed.
Input: Student ID to be deleted.
Output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if student does not exist
* ERROR in case of a database error

ReturnValue addSupervisor (Supervisor supervisor)

Adds a supervisor to the database.
Input: Supervisor to be added.
Output: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters
* ALREADY_EXISTS if a supervisor with the same ID already exists
* ERROR in case of a database error

Supervisor getSupervisorProfile (Integer SupervisorID)

Returns the supervisorwith SupervisorID as its id.

Input: Supervisor id.
Output: The supervisor with SupervisorID if exists. BadSupervisor() otherwise.

ReturnValue deleteSupervisor (Integer SupervisorID)

Deletes a supervisor from the database.
Deleting a supervisor will delete it from <u>everywhere</u> as if it never existed.
Input: Supervisor ID to be deleted.
Output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if supervisor does not exist
* ERROR in case of a database error

You may not use getProfile() functions in your implementation, all must be done via SQL.

3.3 Basic API

ReturnValue studentAttendTest(Integer studentID, Integer testID, Integer semester)

The student with **studentID** is now attending the **test**.
Input: The test the student with **studentID** wishes to attend.
Output: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if student/test does not exist.
* ALREADY_EXISTS if the student already attending the test.
* ERROR in case of a database error

ReturnValue studentWaiveTest(Integer studentID, Integer testID, Integer semester)

The student with **studentID** is no longer attending **test**.
Input: The test that the student with **studentID** wishes to waive.
Output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if student/test does not exist or student does not attend it.
* ERROR in case of a database error

ReturnValue supervisorOverseeTest(Integer supervisorID, Integer testID, Integer semester)

The supervisor with **supervisorID** is now overseeing **test**.
Input: The test the supervisor with **supervisorID** is now overseeing.
Output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if supervisor/test does not exist.
* ALREADY_EXISTS if the supervisor already oversees the test.
* ERROR in case of a database error

ReturnValue supervisorStopsOverseeTest(Integer supervisorID, Integer testID, Integer semester)

The supervisor with **supervisorID** is no longer oversees **test**.
Input: The supervisor with **supervisorID** that is no longer oversees that test**.**
Output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if supervisor/test does not exist or supervisor does not oversee it.
* ERROR in case of a database error

Float averageTestCost()

Returns the average amount of money each test costs.

For example: if there are 2 supervisors oversee test 1 (semester spring) with each salary is 30 and 1 supervisor oversee test 2 (semester winter) for 20, the average is

$$average\ test1 = \frac{30+30}{2} = 30$$

$$average\ test2 = \frac{20}{1} = 20$$

$$return = \frac{30+20}{2} = 25$$

Average is calculated per (TestID, Semester)!
Input: None.
Output:

* The average cost in case of success
* 0 in case of division by 0, -1 in case of other error.

Integer getWage(Integer supervisorID)

Returns the total amount of salaries payed for the supervisor with **supervisorID.**
Input: supervisorID of supervisor.
Output:

* Wage in case of success.
* -1 in case of an error or the supervisor does not exist.

ArrayList<Integer> supervisorOverseeStudent()

Returns an ArrayList of students' IDs that there is a supervisor who oversee the student more than once.
The list should be ordered by IDs in descending order.
Input: None.
Output:

* ArrayList with the students' IDs.
* Empty ArrayList in any other case.

ArrayList<Integer> testsThisSemester(Integer semester)
Returns an ArrayList (up to size 5) of tests' IDs that take place in **semester.**
The list should be ordered by IDs in descending order.
Input: The **semester**.
Output:

* ArrayList with the tests' IDs.
* Empty ArrayList in any other case.

Boolean studentHalfWayThere(Integer studentID)
Returns whether the student with **studentID** <u>has</u> at least (>=) half of the credit points needed in his faculty.

Input: The ID of the student.
Output: whether the student with **studentID** <u>has</u> at least (>=) half of the credit points needed in his faculty. In a case of failure return false.

Integer studentCreditPoints(Integer studentID)
Returns the number of credit points the student with **studentID** has after taking all the tests he/she is currently attending.

Input: The ID of the student.
Output:

> * The number of credit points the student with **studentID** has after taking all the tests he/she is currently attending.
> * 0 in any other case.

Integer getMostPopularTest(String faculty)
Return the test's ID which is the most popular within **faculty**, it has the most students taking it from that **faculty** across all semesters.
In case of equality return the largest ID.

Input: faculty in question.
Output: the test in question, 0 if there is none or in error.

3.4 Advanced API

Note: In any of the following functions, if you are required to return a list of size X but there are less than X results, return a shorter list which contains the relevant results.

ArrayList<Integer> getConflictingTests ()

Returns a list containing conflicting tests' IDs.

Tests are conflicting if and only if they happen on the same day, time and semester.

The list should be ordered by IDs in ascending order.

Input: None

Output:

*ArrayList with the tests' IDs.
*Empty ArrayList in any other case.

ArrayList<Integer> graduateStudents()

Returns a list of up to 5 students' IDs that graduate after adding the current credit points to the tests' credit points they are attending.

The list should be ordered by IDs in ascending order.

Input: None

Output:

*ArrayList with the students' IDs that satisfy the conditions above (if there are less than 5 students, return an Arraylist with the <5 students).
*Empty ArrayList in any other case.

ArrayList<Integer> getCloseStudents (Integer studentID)

Returns a list of the 10 "close students" to the student with id **studentID**.
Close students are defined as students who attend at least (>=) 50% of the tests the student with **studentID** does. Note that one cannot be a close student of himself.

The list should be ordered by IDs in descending order.

Input: The ID of a student.

Output:

*ArrayList with the students' IDs that meet the conditions described above (if there are less than 10 students, return an Arraylist with the <10 students).
*Empty ArrayList in any other case.

Note: students can be close in an empty way (student in question does not attend anything).

4. Database

6.1 Basic Database functions

In addition to the above, you should also implement the following functions:

void createTables()
Creates the tables and views for your solution.

void clearTables()
Clears the tables for your solution (leaves tables in place but without any data).

void dropTables()
Drops the tables and views from the DB.

Make sure to implement them correctly.

6.2 Connecting to the Database using JDBC

Each of you should download, install and run a local PosgtreSQL server from https://www.postgresql.org. You may find the guide provided helpful.

To connect to that server, we have implemented for you the DBConnector class that creates a *Connection* instance that you should work with to interact with the database.

For establishing successful connection with the database, you should provide a proper configuration file to be located under the folder \src\main\resources of the project. A default configuration file has already been provided to you under the name Config.properties. Its content is the following:

```
database=jdbc:postgresql://localhost:5432/cs236363
user=java
password=123456
```

Make sure that port (default: 5432), database name (default: cs236363), username (default: java), and password (default: 123456) are those you specified when setting up the database.

To get the Connection instance, you should invoke the static function DBConnector.*getConnection*(). To submit a query to your database, do the following:

1. Prepare your query by invoking connection.prepareStatement(<your query>). This function returns a PreparedStatement instance.

2. Invoke the function execute() or executeQuery() from the PreparedStatement instance.

The DBConnector class also implements the following functions which you may find helpful:

1. printTableSchemas() – prints the schemas of the tables in the database.

2. printSchema(ResultSet) - prints the schema of the given ResultSet.

3. printResults(ResultSet) - prints the underlying data of the given ResultSet.

6.3 SQL Exceptions

When preparing or executing a query, an SQL Exception might be thrown. It is thus needed to use the try/catch mechanism to handle the exception. For your convenience, the PostgreSQLErrorCodes enum type has been provided to you. It captures the error codes that can be returned by the database due to error or inappropriate use. The codes are listed here:

NOT_NULL_VIOLATION (23502),
FOREIGN_KEY_VIOLATION(23503),
UNIQUE_VIOLATION(23505),
CHECK_VIOLIATION (23514);

To check the returned error code, the following code should be used inside the catch block: (here we check whether the error code *CHECK_VIOLIATION* has been returned)

```
if(Integer.valueOf(e.getSQLState()) ==
PostgreSQLErrorCodes.CHECK_VIOLIATION.getValue())

{

        //do something

}
```

Notice you can print more details about your errors using:

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

Tips

1. Create auxiliary functions that convert a record of ResultSet to an instance of the corresponding business object.

2. Use the enum type PostgreSQLErrorCodes. It is highly recommended to use the exceptions mechanism to validate Input, rather than use Java's "if else".

3. Devise a convenient database design for you to work with.

4. Before you start programming, think which Views you should define to avoid code duplication and make your queries readable and maintainable.

   (Think which sub-queries appear in multiple queries).

5. Use the constraints mechanisms taught in class to maintain a consistent database.
   Use the enum type PostgreSQLErrorCodes in case of violation of the given constraints.

6. Remember - you are also graded on your database design (tables, views).

7. Please review and run example.java for additional information (ArrayList, String) and implementation methods.

8. AGAIN, USE VIEWS!

Submission

Please submit the following:

A zip file named <id1>-<id2>.zip (for example 123456789-987654321.zip) that contains the following files:

1. The file Solution.java where all your code should be written in.

2. The file <id1>_<id2>.pdf in which you explain in detail your database design and the implantation of the API. Is it **NOT** required to draw a formal ERD but it is indeed important to explain every design decision and it is highly recommended to include a draw of the design (again, it is **NOT** required to draw a formal ERD).

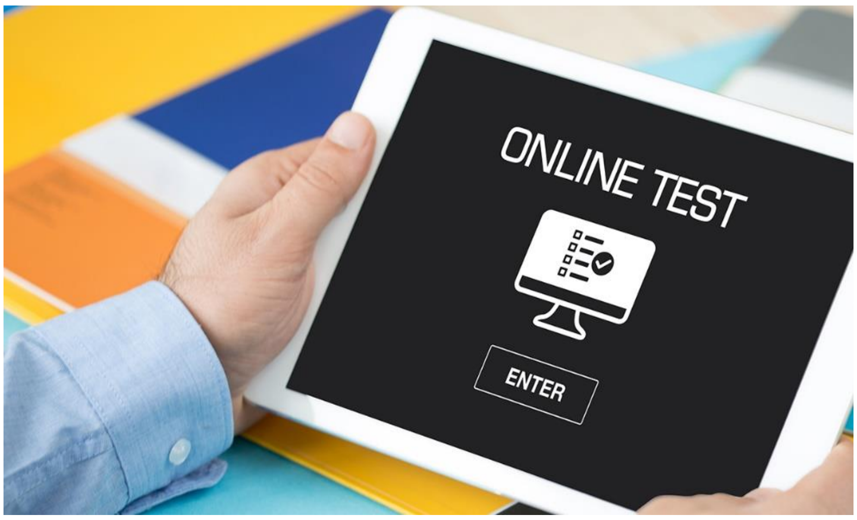
3. The file <id1>_<id2>.txt with nothing inside.

Make sure that is the exact content of the zip with no extra files/directories and no typos.

Note that you can use the unit tests framework (Junit) as explained in detail in the PDF about installing IDE, but no unit test should be submitted.

Any other type of submission will fail the automated tests and result in 0 on the wet part, which is 50% of the total grade (your code will also go through dry exam).

You will not have an option to resubmit in that case!



Good Luck!