Practical Work: Fundamental Java Programming – fall 2022 students

The Goal of this practical work is to assess the student's ability to write and complete Java code, to show his operational skills in real situation.

This Practical Work is composed of 3 domains:

1. UML & OOP Approach
2. Business Logic Implementation using Java
3. Data Access with Java

Each domain needs to be completed to go on, as those domains are ordered from the most general to the most specific.

Important note:

You'll have to deliver all the projects created for that practical workshop; you must place all of them in the same folder.

Remark: You will have a global bonus If you follow good coding practices, like putting comments on critical code, avoid duplications, enforcing naming conventions, etc.

In the home/subject folder, you will have access to the 4 csv files necessary to complete this practical work

- patients.csv
- medications.csv
- insurances.csv
- prescriptions.csv

Read carefully all this document before starting, it's important to have a good overview of this exam! Do your best to implement each exercise correctly before moving to the next one.

Final format for your delivery (replace ${id} by your epita user name, replace "your-packages" and

"your-classes" by your own code.

```
/home/submission
        +- ${id}-fund-java
                +- output
                        +- model.png
                +- src/main
                        +- (your-packages).(your classes)
                +- src/test
                        +- TestOOP1.java
                        +- TestOOP2.java
                        +- TestBLI1.java
                        +- TestBLI2.java
                        +- TestJDB1.java
                        +- TestJDB2.java
                        +- TestJDB3.java
                        +- TestJDB4.java
            +- resources
                        +- patients.csv
                        +- medications.csv
                        +- … (other CSVs, .sql files)
```

Domain 1: OOP & UML, file handling in Java

Exercise OOP1

5 minutes **1ts**

🎯 *Create a runnable program in Java*

Create a new Java project in IntelliJ named "${id}-fund-java", id being your epita user name. Create an executable class `TestOOP1` in a *.launcher package (replace the star in the package name by an appropriate name).

Make this class output ${id} (check the important note at the beginning of the subject, to understand what ${id} should be replaced with).

Exercise OOP2

25 minutes **5pts**

🎯 *Read data from a formatted file and deserialize them in the appropriate data structure using Java*

a. Consider the following datasets (respectively patients.csv and insurances.csv):

```
pat_num_HC;pat_lastname;pat_firstname;pat_address;pat_tel;pat_insurance_id;pat_sub_date

"1256987452365";Martin;Bernard;Chatillon;"0106060606";2;01/10/2010

"1852458963215";Chalme;Antoine;Paris;"0105050505";1;01/01/2017

"1985236548520";Daulne;Paul;Puteaux;"0107070707";3;01/05/2008

"2365987542365";Solti;Anna;Montrouge;"0108080808";4;01/10/2010

"2658954875210";Dart;Pauline;Bourg la reine;"0109090909";4;01/01/2015

"2758965423102";Chalme;Julie;Paris;"0105050505";1;01/06/2017
```

Patients

```
insurance_id;insurance_name

1;MACIF

2;MGEN

3;MAAF

4;ADREA

5;APICIL
```

Insurances

a. Create `Patient` and `Insurance` classes, according to the data and types shown in the demonstrated CSV files, *pat_sub_date* is the date the patient has subscribed to his insurance.

b. Copy paste the 2 datasets from the *subject* folder.
c. Create a `PatientReader` class and a `InsuranceReader` class that can read (through a function `readAll()`) a list of respective object types (Patient and Insurance) from the 2 files containing the 2 previous datasets.
d. Create a class `TestOOP2`, containing a `main(String[] args)` which will call those 2

classes and display the list of instances (think of implementing the `toString()` method).

Exercise OOP3 (bonus)

UML Class Diagram design

a. Design a uml class diagram "output/model.png" that represents your model.

Domain 2. Business Logic Implementation and Data Structures

For this domain, you must implement a service class that will provide information for a potential UI. Most of the information returned by the service functions are projections or calculation based on existing data, to give useful information for the user, without persisting them in the database. The first function example is shown after, in the BLI1 exercise, where you must compute the seniority of a patient based on its subscription date.

Exercise BLI1

*10 minutes **1pts***

🎯 *Create services to delegate Business Logic implementation*

    a. Create a service class named "PatientBLService" (standing for Patient Business Logic Service) that can compute the seniority of patients. The method should have the following prototype:

```java
public static Integer computeSeniority(Patient patient){
        // method body
}
```

It should take the subscription date in the dataset to compute the patient seniority (as of the current date).

    b. Create an executable class TestBLI1 implementing the following scenario:
- It will use the PatientReader to load a list of Patient from a the csv file
- It will take the 7th patient entry in the list (reading order) and output its seniority in the console

Exercise BLI2

*30 minutes **5pts***

🎯 *Choose the appropriate Java Data Structure to solve a Business Logic requirement*

    a. In the same service class (PatientBLService), create a method computing the seniority by patient. The returned data structure should be able to store the association between the seniority and the associated patient identifier (*pat_num_HC*). Here is the expected prototype

```java
public static <ret_type> computeSeniorityByPatient(List<Patient>
patients){
        // method body
}
```

The expected structure should look like this (new lines have been inserted between each entry) :

```
{
    1256987452365=10,
    1852458963215=4,
     …
}
```

Notice again: the patient is here represented by his *pat_num_HC*.

b. Create an executable class named TestBLI2 which implements the following scenario in its main method:
    o   Load a list of patients thanks to the PatientsReader
    o   Use the computeSeniorityByPatient() method to get the appropriate data structure that associates a patient to his seniority
    o   Display the data structure in the console.

Domain 3. Data Access, implementation with Java and H2

Exercise JDB1

10 minutes ***1pts***

🎯 *Run a statement against a database using Java*

a. Create a class `TestJDB1` with a `test()` method that will be able to connect to an in-memory embedded h2 database (driver provided in the assignment or as link in the help below)

b. Complete the `test()` method by preparing a statement that will create the table "`PATIENTS`" with the appropriate columns names, types, and constraints (you can add an auto_increment feature on a new "id" column, or make this column use the type IDENTITY)

c. Invoke that test() method in the main method of your launcher class

```
// Previous code

TestJDB1.test() ;
```

> **Hints**
> -   To setup the connection, you need to include the h2 jdbc driver as seen during lectures.
> -   The jar can be downloaded from here
> -   To open a connection in an in-memory h2 database, one has to connect to this url:
>
> **jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1**
>
> -   **Important** : remember that the in-memory db will lose all the data between two connections, so beware to reuse the same connection or you will get unexpected behavior.

Exercise JDB2

*25 minutes 3**pts***

🎯 *Define a strategy to handle data access and implement it using Java*

a. Create a Data Access Object (DAO) class named `PatientDAO` in the appropriate package. This DAO will have the 4 main actions to handle patients-related data:
    o Create a patient,
    o Update a patient,
    o Delete a patient,
    o Search patients using search criteria

b. Create a class named TestJDB2 implementing the following scenario:
   1. Use the PatientReader to read from the CSV file and get a List of Patient instances
   2. Using the create() action of an instance of the PatientDAO class, save the list of Patient instances in the database.
   3. Search a precise patient using first name and lastname (no search by id!)
   4. Update whatever patient you have found with your search
   5. Delete this patient from the database using the delete() method.

Exercise JDB3

*15 minutes 2**pts***

🎯 *Implement a complete PDM (Physical Data Model) in Java*

a. Consider the following additional data:

```
medication_code;medication_name;medication_comment

1;Advil;anti-inflammatory

2;Doliprane;paracetamol

3;Spasfon;antispasmodic

4;Smecta;gastric dressing

5;Strepsil;antiseptic
```

Medication

```
presc_id;presc_ref_pat;presc_code;presc_days

1;2758965423102;1;2

2;1985236548520;2;4

3;1852458963215;1;10

4;2365987542365;4;5

5;1256987452365;4;5

6;2758965423102;1;2
```

Prescription

b. Create the Medication and Prescription classes, with the according data types,
c. Create CSV readers for those 2 classes.
d. Create SQL scripts to create the tables INSURANCES, MEDICATIONS and PRESCRIPTIONS, and store them respectively in the create-insurances.sql, create-medications.sql and create-prescriptions.sql. Put your creation script for patients (JDB1) at the same format in the create-patients.sql file.
e. Create a class TestJDB3 that will load run the 4 scripts from respective files to initialize the DB.

Exercise JDB4

*30 minutes **2pts***

🎯   *Setup a complete data access layer in Java to match business requirements*

    a. Create 3 classes, containing 4 methods (search/update/delete/create), like the class done in JDB2

-        `InsuranceDAO`
-        `MedicationDAO`
-        `PrescriptionDAO`

    b. Reduce the code as much as possible, using either aggregation or inheritance,

    c. Create a test class named `TestJDB4` that will display the name of each patient with his associated treatment (*medication_name*) and the number of days (*presc_days*) this prescription is valid for. The many to one relationship between `MEDICATIONS` and `PRESCRIPTIONS` can be done through the *presc_code* column. Between `PATIENTS` and `PRESCRIPTIONS` : *presc_ref_pat*