Java Practical Work: Fundamentals

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 4 domains

- Data modeling
- Serialization / Deserialization
- Business Logic Implementation
- Toward Data Science

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

For each resource which is asked to be created, you have to replace **${id}** by your epita login, replacing the “.” (dot) by a “_“(underscore).

Example : if your login is **jean-paul.grandclair**, your ${id} will become : **jean-paul_grandclair**

You will have to deliver all the projects created for that practical workshop, you must place all of them in the same folder, the folder has to be placed under “submission”

Remark: you will have a global bonus If you follow good coding practices, like putting comments on critical code, using loggers, Javadoc, code formatting, naming conventions, code organization…

Remark : all the ***`Test`**** classes defined later in this document will have a method “***`public static void test()`***”.

Domain 1: Data Modeling (15 minutes - 3 points)

Exercise DMO1

*5 minutes **1pt***

Create a new Java Project that contains runnable class in the appropriate package. This class should be named “**Launcher**”. It should contain a method with a specific signature allowing Java to launch this class as an entrypoint of your program.

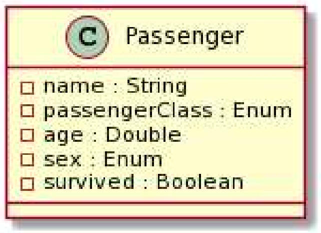
```
public class Launcher {
      public static void ${methodName}(${arguments}){
            TestDMO2.test();
            // TestSER1.test();
            // and so on
       }
}
```

Change the ${methodName} and ${arguments} by the appropriate values.

Exercise DMO2

*10 minutes **2pt***

a. Create a new Java Class named “Passenger” that will represent the characteristics of a titanic passenger. Hereafter is the UML modeling of this passenger class.



Remarks:

- The Sex attribute can have 2 values (“male” or “female”),
- The survived attribute would take the value true if the passenger survived
- The passengerClass attribute can take 3 values ("1st", "2nd", "3rd").

b. Create the appropriate constructor(s) and handle encapsulation.

c. Add a **toString()** method that could return the following kind of output when called on an instance:

```
"Passenger [name="Abbing, Mr Anthony", survived="true"]";
```

d. Create a Test class (no Junit) named **TestDMO2** class that will create an instance of Person that will produce the previous output in the console
Place the test execution to the main execution in the Launcher class by adding the following line to the main method:

```
// Previous code

TestDMO2.test() ;
```

Domain 2: Deserialization / Serialization (55 minutes – 6 points)

Exercise SER1

*20 minutes **2pts***

a. Consider the following text file provided in the “subject” folder (data.csv)

```
Name                                                    ;PClass;Age ;Sex   ;Survived
Allen, Miss Elisabeth Walton                            ;1st   ;29  ;female;1
Allison, Miss Helen Loraine                             ;1st   ;2   ;female;0
Allison, Mr Hudson Joshua Creighton                     ;1st   ;30  ;male  ;0
Allison, Mrs Hudson JC (Bessie Waldo Daniels)           ;1st   ;25  ;female;0
Allison, Master Hudson Trevor                           ;1st   ;0.92;male  ;1
Anderson, Mr Harry                                      ;1st   ;47  ;male  ;1
Andrews, Miss Kornelia Theodosia                        ;1st   ;63  ;female;1
Andrews, Mr Thomas, jr                                  ;1st   ;39  ;male  ;0
Appleton, Mrs Edward Dale (Charlotte Lamson)            ;1st   ;58  ;female;1
Artagaveytia, Mr Ramon                                  ;1st   ;71  ;male  ;0
Astor, Colonel John Jacob                               ;1st   ;47  ;male  ;0
Astor, Mrs John Jacob (Madeleine Talmadge Force)        ;1st   ;19  ;female;1
Aubert, Mrs Leontine Pauline                            ;1st   ;    ;female;1
Barkworth, Mr Algernon H                                ;1st   ;    ;male  ;1
Baumann, Mr John D                                      ;1st   ;    ;male  ;0
Baxter, Mrs James (Helene DeLaudeniere Chaput)          ;1st   ;50  ;female;1
Baxter, Mr Quigg Edmond                                 ;1st   ;24  ;male  ;0
```

b. Create a Test class named “TestSER1” that contains a test() method, that will read all the lines from the file and will display the 2nd line of the file in the console.

c. Add the test execution to the main method in the launcher class as the following

```
        // Previous code

        TestSER1.test() ;
```

Hints

- You can use the ***split(“,”)*** method on each line to extract the sub parts of the csv line
- You can use the ***trim()*** or ***strip()*** method on each part to eliminate the unnecessary blank spaces (don’t forget to work on the result of this call, the original variable is left untouched)

Exercise SER2

*20 minutes **2pts***

a. Consider the same data.csv file

b. Create a ***PassengerCSVDAO*** class in the appropriate package and create a ***readAll()*** method in it. Inspire from the ***TestSER1*** test method code and place it under the ***readAll()*** method. The ***readAll()*** method should return a list of Person instances, sorted by their height.
c. Create a ***TestSER2*** class with a ***test()*** method that will invoke the ***PassengerCSVDAO.readAll()*** instance and that will display it in the console.
d. Add the test execution to the main method in the launcher class as the following

```
        // Previous code
        TestSER2.test() ;
```

Hints
You can use ***Collections.sort(<your_list>)*** to sort the List of ***Passenger***, or perform the sorting by "manually" implementing it.

Exercise SER3

15 minutes ***2pts***

a. Consider the previous ***PassengerCDVDAO*** class.
b. Create a method "***writeAll(List<Passenger> passengers)***" in that class that will produce the same CSV file but with different Column order like the following.
c. The output should be placed in **data_output.csv**

The order of the columns was :

```
Name      ;PClass;Age ;Sex   ;Survived
```

And should now be :

```
PClass    ;Name  ;Sex ;Age   ;Survived
```

d. Create a test class TestSER3 that will invoke the ***readAll()*** method to get a List<Person> variable, and then use the ***writeAll()*** method to write this as an output.
e. Add the test execution to the main method in the Launcher class as follows

```
        // Previous code
        TestSER3.test() ;
```

Domain 3. Business Logic Implementation (35 minutes – 4 points)

For that domain, if you were not successful to read from the csv file (domain 2 exercises), you can consider working on a "handmade" list of Persons.

Exercise BLI1

*30 minutes **4pts***

a. Write a class *PassengerDataService* that will perform some statistics / filtering on the persons data through the following methods:

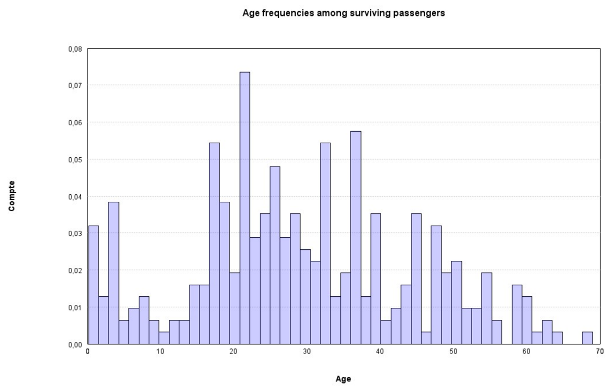
   1. *List<Passenger> filterSurvived(List<Person> passengers, Boolean survived)*
      Returns the list of passengers who have survived.

   2. *int averageAge(List<Passenger> passengers)*
      Calculate the average age of the given passengers list. If the age is missing, then the passenger age should be ignored and not taken in account for calculation.

   3. *Map<String, Double> calculateAgeDistribution(List<Passenger> passengers)*
      Returns the distribution by age of the passengers list given as an argument. The age is rounded by taking the whole part of the age characteristic. The result can be represented as follows:

```
{
        0 => 0.002
        1 => 0.005
        ..
        30 => 0.15
        …
}
```

      The left-hand side is the age and the right-hand side is the percentage of the total population that shares this age. This would allow to draw this kind of graph



b. Write a class named *TestBLI1* with a *test()* method that will contain the following use cases :

   1. Call to the *filterSurvived()* on the persons list true as a threshold, with output of the list size in the console.

2. Call to the ***averageAge()*** function on the passengers list that resulted from the previous filter application, output the result in the console.
3. Call ***calculateAgeDistribution()*** on the passengers list, you should proceed with an output in the console that looks like :

```
{0=5, 1=5, 2=4, 3=6, 4=6, 5=2, 6=2, 7=1, 8=4, 9=2, 11=1, 12=2, 13=2, 14=1, 15=4 …
```

c. Invoke that ***test()*** method in the main method of your launcher class by adding the following line:

```
// Previous code
TestBLI1.test() ;
```

Domain 4. Implementing a statistical classifier (~ 60 minutes, 7pts)

The goal of this domain is to put all your knowledge to realize the implementation of a statistical classifier to predict if passenger survives or not. It identifies 2 groups: Survivors and Deceased.

This classifier is called a centroid classifier because it identifies the most common characteristics of each cluster and form an “ideal” representant of each cluster by calculating the mean of each characteristics.

Exercise TDS1

20 minutes, 2pts.

Create a class named ***TestTDS1***, containing a ***test()*** method. In this method:

a. Calculate the passengers distribution on the PClass characteristic and then calculate the weighted mean on Age and PClass and Sex for each group
b. Then create 2 instances, 1 for each group assigned with the respective calculated values. The two instances are the centroids of each group
c. Create a static method ***getSurvivedCentroids(List<Passenger> list)*** in TestDS1 that returns a ***Map<Integer, Passengers***> which associates the class label (Survived characteristic value) to each centroid.
d. Produce the following output in the ***test()*** method

```
Typical surviving passenger : PClass = ${pclass}, Sex = ${sex}, Age = ${age}

Typical not surviving passenger : PClass = ${pclass}, Sex = ${sex}, Age = ${age}
```

e. Add this call to the test method in the launcher

```
        // Previous code
        TestTDS1.test() ;
```

Exercise TDS2 (non guided)

*30 minutes, **2,5pts***

Create a class TDS2 with a method test that should produce that will add a “predicted” column in the original table. The value of this column will be the survival prediction for each line.

The prediction will be made using Euclidian distance between each centroid (obtained from the previous exercise) and the current occurrence. The closest distance between the centroid and the current occurrence will predict the occurrence to belong the centroid group.

Produce a csv file named “***predictions.csv***” containing that column when calling

```
        // Previous code
        TestTDS2.test() ;
```

From the ***Launcher*** class.

Exercise TDS3 (non guided)

*30 minutes, **2,5 pts***

Based on the previous predictions, one can evaluate the performance of this classifier by identifying the confusion matrix.

A confusion matrix is identifying for each class the number of false positive, false negative, true positives and true negatives.

- True positives are the occurrences that are predicted survivors and have survived
- True negative are the occurrences that are predicted deceased and have actually not survived
- False negative: occurrences predicted deceased but were survivors
- False positive: occurrences predicted survivors but were deceased.

Predicted ↓ / Actual →	Survives	deceases
Survives	Tp	Fp
Deceases	Fn	Tn

Example :

If we have 9 survivors and 9 deceased passengers, but predicted 7 survivors and 11 deceased, the confusion matrix could be like the following:

Predicted ↓ / Actual →	Survives	Deceases
Survives	6	1
Deceases	3	8

Calculate and display the confusion matrix by calling the method

```
        // Previous code

        TestTDS3.test() ;
```

From the ***Launcher*** class, it should produce this output.

```
        Confusion matrix : [[6,1],[3,8]]
```