Cours MOOC EPFL d'introduction à la programmation orientée objet, illustré en Java

Fourth assignment (graded) : Polymorphism

J. Sam & J.-C. Chappelier

This assignment consists of two exercises to submit.

1 Exercise 1 — Travel agency

A travel agent would like you to you help him handle his travel offers.

1.1 Description

Download the source code available at the course webpage[1] and complete it.

WARNING: you should modify neither the beginning nor the end of the program, only add your own lines as indicated. It is therefore very important to respect the following procedure (the points 1 and 3 concern only Eclipse users):

1. remove automated formatting of the code in Eclipse:
   `Window > Preferences > Java > Editor > Save Actions`
   (and untick the formatting option if it's on);
2. save the downloaded file as `Voyage.java` (respect the upper case). If you work with Eclipse, save it to `[projectFolderUsedForThatExercise]/src/`;
3. refresh the Eclipse project where the file is stored (right click on the project > "refresh") in order to take into account the new file;

[1] `https://d396qusza40orc.cloudfront.net/intropoojava/assignments-data/Voyage.java`

4. write your code between these two provided comments:

```
/*****************************************
 * Completez le programme a partir d'ici.
 *****************************************/

/*****************************************
 * Ne rien modifier apres cette ligne.
 *****************************************/
```

5. save and test your program to be sure that it works properly, for example using the values given below;
6. upload the modified file (still named `Voyage.java`) in "OUTPUT submission" (not in "Additional"!).

1.2 The code to be produced

The travel options Our travel agent sells traveling kits composed of different *options*.

You first task consists in implementing a class `OptionVoyage` to represent such options.

An *option* (`OptionVoyage` class) is characterized by:

- its *name*, a string;
- its *fixed price* (a `double`).

The `OptionVoyage` class includes:

- a constructor initializing the attributes using values passed as parameters in an order which is compatible with the given `main`;
- a method `getNom` returning the name of the option;
- a method `double prix()` returning the fixed price for the option;

- a method `toString` producing a representation of the option in the form of a string, according to the following format:

  `<nom> -> <prix> CHF`

  where `<nom>` is the name of the option and `<prix>` is its price.

You are asked to implement the class `OptionVoyage`. You must respect a good encapsulation.

This part of your program can be tested using the code given between `// TEST 1` and `// FIN TEST 1`.

The traveling options can be of course specialized using different sub-classes. You will now have to model two of them: the means of transport (`Transport` class) and the accommodation during the trip (`Sejour` (Lodgement) class).

The `Sejour` class An instance of `Sejour` will be characterized by the *number of nights* (an integer) and the *price per night* (a double).

The price of a lodgement is simply the number of nights multiplied by the price per night, to which we add the fixed price of the option.

The `Transport` class An instance of `Transport` will be characterized by a boolean indicating whether the trip is long.

The price of the transport is the constant `TARIF_LONG` (`1500.0`) if the trip is long and `TARIF_BASE` (`200.0`) otherwise, to which we add the fixed price of the option. Constants will be declared as `public`.

You should now make sure that the class `OptionVoyage` is specialized in two sub-classes: `Transport` and `Sejour` fitting the above description.

The hierarchy of classes will have:

- constructors conforming to the provided `main`. The arguments for `Transport` will be given according to the following order : the name of the option, the fixed price and a boolean with value `true` for long trips and `false` otherwise. The arguments for `Sejour` will be : the name, the fixed price, the number of nights and the price per night.

By default, a `Transport` has a short trip.

- specific redefinitions of the method `prix`. These specializations should contain no
code duplication at all and will be usable in a polymorphic way.

This part of your program can be tested using the code included between `// TEST 2` and `// FIN TEST 2`.

Travel kit The travel agent sells kits consisting of multiple options.

You are asked to code a class `KitVoyage` as a « heterogeneous collection » of `OptionVoyage` (an `ArrayList`).

The class `KitVoyage` will also be characterized by the *departure* and the *destination* of the kit (two `String`).

The class `KitVoyage` will have:

- a constructor compatible with the provided `main` (see the portion of the code between `// TEST 3` and `// FIN TEST 3`);

- a method `double prix()` that will calculate the price of the kit as the sum of the prices of all the options;

- a method `toString`, compatible with the provided `main` generating a `String` representation of the kit according to the following format:

```
Voyage de <depart> à <destination>, avec pour options :
- <nom option1> -> <prix option1> CHF
- ....
- <nom optionN> -> <prix optionN> CHF
Prix total : <prix du kit> CHF
```

where `<depart>` is the departure of the kit, `destination`, its destination and `<prix du kit>` its price. The built string will end with `\n`.

- a method `ajouterOption`, compatible with the provided `main` and allowing to add an `OptionVoyage` to the collection of options of the kit (the options will be added at the end of the collection). If the argument passed to `ajouterOption` is `null`, the collection will remain unchanged (nothing added);

- a method `annuler()` that resets the collection to an empty set (use the `clear` method of `ArrayList`);

- a method `getNbOptions` returning the number of travel options of the kit.

This part of the program can be tested using the portion of the code included between `// TEST 3` and `// FIN TEST 3`.

1.3 Execution example

```
Test partie 1 :
----------------
Séjour au camping ->  40.0 CHF
Visite guidée : London by night ->  50.0 CHF

Test partie 2 :
----------------
Trajet en car  ->  250.0 CHF
Croisière ->  1500.0 CHF
Camping les flots bleus ->  320.0 CHF

Test partie 3 :
----------------
Voyage de Zurich à Paris, avec pour options :
   - Trajet en train ->  250.0 CHF
   - Hotel 3* : Les amandiers  ->  540.0 CHF
Prix total : 790.0 CHF


Voyage de Zurich à New York, avec pour options :
   - Trajet en avion ->  1550.0 CHF
   - Hotel 4* : Ambassador Plazza   ->  600.0 CHF
Prix total : 2150.0 CHF
```

cd

2 Exercise 2 — Rally

An organizer of motorcycle and car rallies asks your help for organizing his races.

2.1 Description

Download the source code available at the course webpage[2] and complete it.

WARNING: you should modify neither the beginning nor the end of the program, only add your own lines as indicated. It is therefore very important to respect the following procedure (the points 1 and 3 concern only Eclipse users):

1. remove automated formatting of the code in Eclipse:
   `Window > Preferences > Java > Editor > Save Actions` (and untick the formatting option if it's on);
2. save the downloaded file as `Course.java` (respect the upper case). If you work with Eclipse, save it to `[projectFolderUsedForThatExercise]/src/`;
3. refresh the Eclipse project where the file is stored (right click on the project > "refresh") in order to take into account the new file;
4. write your code between these two provided comments:

```
/*****************************************
 * Completez le programme a partir d'ici.
 *****************************************/

/*****************************************
 * Ne rien modifier apres cette ligne.
 *****************************************/
```

5. save and test your program to be sure that it works properly, for example using the values given below;
6. upload the modified file (still named `Course.java`) in "OUTPUT submission" (not in "Additional"!).

[2] `https://d396qusza40orc.cloudfront.net/intropoojava/assignments-data/Course.java`

2.2 The code to be produced

You are asked to complete the code according to the description that follows.

1) The `Vehicule` class You should firstly implement a class `Vehicule` that allows to represent a vehicle that participates in the races.

A *véhicule* (vehicle) is characterized by:

- its *name*, a string like « Ferrari » for example ;
- its *maximal speed* (a `double`) ;
- its *weight* in kg (an `int`) ;
- and the level of *fuel* in the tank (an integer).

The class `Vehicule` will include :

- a constructor, compatible with the provided `main` method, initializing the attributes using values passed as parameters and a default constructor initializing the name with *"Anonyme"*, the level of the fuel with zero, the maximal speed with 130 and the weight with 1000 ;
- a method `toString` producing a `String` containing all the characteristics of the vehicle except for the level of fuel, respecting ***strictly*** the following format :
  `<nom> -> vitesse max = <vitesse max> km/h, poids = <poids> kg`
  where `<nom>` is the name of the vehicle, `<vitesse max>` is its maximal speed and `<poids>`, its weight ;
- a method `meilleur(Vehicule autreVehicule)` returning `true` if the current instance has a better performance than the `autreVehicule` ;
- the « getters » `getNom()` (gets the name), `getVitesseMax()` (gets the maximal speed), `getPoids()` (gets the weight) and `getCarburant()` (gets the level of fuel).

Finally, this class will include and use a tool method `double performance()`. This method must return an estimation of the performance of the vehicle like the

ratio between the maximal speed and its weight (the lighter and the faster the vehicle is, the better its performance because it consumes less energy) ;

You are asked to implement the class `Vehicule` respecting a good encapsulation.

This part of you program can be tested by the portion of the provided `main` method contained between `// TEST 1` and `// FIN TEST 1` (see the given code).

2) Cars and motorcycles The vehicles participating in the rallies can be either cars or motorcycles.

A car (class `Voiture`) is characterized by additional information indicating its category (« course » or « tourisme »).

A motorcycle (class `Moto`) is characterized by a boolean indicating if it has a *Sidecar*.

Program now the hierarchy of classes allowing you to represent these two types of vehicles by providing:

- constructors conforming to the provided `main` (in the portion of the code between `// TEST 2` and `// FIN TEST 2` ; **by default, a `Moto` does not have a Sidecar** ; it is not necessary to test the validity of the value of the arguments in the constructors ;
- redefinitions specific to the method `toString` ; these specializations won't contain any code duplication.

By the way:

- the representation of a car in the format of a `String` will respect ***strictly*** the following format :
  `<nom> -> vitesse max = <vitesse max> km/h, poids = <poids> kg, Voiture de <categorie>`
  where `<nom>` is the name of the vehicle, `<vitesse max>` its maximal speed, `<poids>`, its weight and `<categorie>`, its category ;
- the representation of a motorcycle in the format of a `String` will respect ***strictly*** the following format :

`<nom> -> vitesse max = <vitesse max> km/h, poids = <poids> kg, Moto, avec sidecar`
if it has a « Sidecar », or if not:
`<nom> -> vitesse max = <vitesse max> km/h, poids = <poids> kg, Moto`
with `<nom>` the name of the vehicle, `<vitesse max>` its maximal speed, and `<poids>`, its weight.

The class `Voiture` will also contain the method `getCategorie()`.

This part of your program can be tested by the portion of the given `main` method contained between `// TEST 2` and `// FIN TEST 2` (see the provided code).

3) the classes `GrandPrix` and `Rallye` You are now asked to code a class `GrandPrix` as a « heterogeneous collection » of vehicles. This collection represents the set of the vehicles participating in a race. It is modeled using an `ArrayList`.

This class inherits from a class `Rallye` which contains uniquely one method `boolean check()`. This method must allow to verify if the vehicles have the right to race together. The method `check` **cannot be concretely defined in the class `Rallye`**.

The class `GrandPrix` will have:

- a method `ajouter` allowing to add a vehicle in the set of participants (the insertion will be done in the end of the collection) ; this method will conform to the provided `main` method (in the portion of the code between `// TEST 3` and `// FIN TEST 3`) ;
- for a rally of the type `GrandPrix` the cars don't have the right to race with two-wheeled; the motorcycles having a «Sidecar» **are not considered as vehicles with two wheels** ; the two-wheeled have the right to race together.

In order to test the compatibility of the vehicles, you will add to the **hierarchy** of `Vehicule` a method:

```
boolean estDeuxRoues()
```

returning `true` when the vehicle is of type two wheels and `false` in the contrary case. You will consider that **a basic vehicle is not a two-wheeled**.

This part of your program can be tested by the portion of the provided `main` method contained between `// TEST 3` and `// FIN TEST 3` (see the provided code).

4) The race is launched Complete the class `GrandPrix` by adding to it a method `void run(int tours)` simulating the progress of the race according to the following algorithm:

- start by testing if the vehicles have the right to race together ; the message *"Pas de Grand Prix"* followed by an End Of Line will be displayed if not and the method `run` should terminate its execution ;
- when the race takes place : for every vehicle, deduce as much fuel as the `tours` ; only the vehicles that still have fuel (> 0) arrive to the finishing line ;
- among the vehicles that reach the finishing line, select the most efficient one (the one which is better than all the others) and display it respecting ***strictly*** the following format :

  ```
  Le gagnant du grand prix est :
  <representation>
  ```

  where `<representation>` is the representation of the winning vehicle in the format of a `String`, as it is produced by `toString`.

  If no vehicle reaches the starting line, display the message

  ```
  Elimination de tous les vehicules
  ```

  followed by an End Of Line.

This part of your program can be tested by the portion of the provided `main` method contained between `// TEST 4` and `// FIN TEST 4` (see the provided code).

2.3 Execution example

```
Test partie 1 :
----------------
Anonyme -> vitesse max = 130.0 km/h, poids = 1000 kg

Ferrari -> vitesse max = 300.0 km/h, poids = 800 kg

Renault Clio -> vitesse max = 180.0 km/h, poids = 1000 kg
Le premier vehicule est meilleur que le second

Test partie 2 :
```

```
----------------
Honda -> vitesse max = 200.0 km/h, poids = 250 kg, Moto, avec sidecar

Kawasaki -> vitesse max = 280.0 km/h, poids = 180 kg, Moto

Lamborghini -> vitesse max = 320.0 km/h, poids = 1200 kg, Voiture de course

BMW -> vitesse max = 190.0 km/h, poids = 2000 kg, Voiture de tourisme


Test partie 3 :
----------------
true
false
true
false

Test partie 4 :
----------------
Premiere course :
Le gagnant du grand prix est :
Lamborghini -> vitesse max = 320.0 km/h, poids = 1200 kg, Voiture de course

Deuxieme  course :
Elimination de tous les vehicules

Troisieme  course :
Pas de Grand Prix
```