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

Third assignment (graded) : Inheritance

J. Sam & J.-C. Chappelier

This assignment consists of two exercises to submit.

1 Exercise 1 — Philately

A philatelist wishes to estimate at which price he could sell his stamps. The purpose of this exercise is to write a program which allows him to do so.

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 `Philatelie.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/Philatelie.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 `Philatelie.java`) in "OUTPUT submission" (not in "Additional"!).

The provided code does the following :

- creates a collection of stamps (rare or commemorative) ;
- and displays the price of every stamp of this collection.

The hierarchy of `Timbre` (Stamp in french) is missing and it is what we ask you to write.

A stamp is characterized by : its *code*, its *année d'émission* (issue year), its *pays d'origine* (country of origin) and its *valeur faciale* (denomination value) in francs (the equivalent in francs of the value printed on the stamp).

Our philatelist distinguishes two broad categories of stamps, which are distinguished by the way the selling price is computed.

- the *rare* stamps (class `Rare`) : possesses an extra attribute indicating the number of copies that are inventoried throughout the world ;
- the *commemorative* stamps (`Commemoratif`) : without any particular specific attribute.

Selling price of the stamps The selling price of a *timbre* (stamp) of any kind is its denomination value, if the stamp is less than 5 years old. If not, the selling price is the denomination value multiplied by the age of the stamp and by the coefficient 2.5.

The selling price of a *rare* stamp starts from a *prix de base* (base price) : 600 francs if the number of the inventoried copies is less than 100, 400 francs if the number of copies is between 100 and 1000 and 50 francs in any other case. The selling price of a rare stamp is then given by the formula `prix_base * (age_timbre / 10.0)`. **The different constants involved in the calculation are given in the provided file**.

The selling price of a *Commemoratif* stamp is twice as much as the price of a generic one.

Basic stamp The public methos of the class `Timbre` are :

- constructors that conform to the provided `main()` method, with the following order of the parameters : the code, the issue year, the country of origin, and the denomination value ; each one of this parameters takes a default value : `VALEUR_TIMBRE_DEFAUT` for the denomination value, `PAYS_DEFAUT` for the country, `ANNEE_COURANTE` for the year and `CODE_DEFAUT` for the code ; the construction of a stamp can be thus done with zero, one, two, three or four parameters (respecting the order given above) ;
- a method `vente()` returning the selling price in the format of a `double` ;
- a method `toString()` producing a representation of a `Timbre` respecting ***strictly*** the following format :
  ```
  Timbre de code <code> datant de <annee> (provenance <pays>)
   ayant pour valeur faciale <valeur faciale> francs
  ```
  in only one line where `<code>` is to be replaced by the code of the stamp, `<annee>` by its issue year, `<pays>`, by its country of origin and `<valeur faciale>`, by its denomination value ;
- the method `age()` returns the age of a stamp in the format of an `int` (difference between `ANNEE_COURANTE` (current year) and the issue year of the stamp) ;
- the getters `getCode()`, `getAnnee()`, `getPays()` and `getValeurFaciale()`.

Rare stamps The class `Rare` has to provide as public methods :

- constructors that conform to the provided `main()`, with the following order for the parameters : the code, the issue year, the country of origin, the denomination value and the number of inventoried copies. The number of copies should always be provided when the constructors are called. The other parameters, if they are not provided, have the default values that are indicated for the basic stamps ;

- a getter `getExemplaires()` ;

- a method `toString()` producing a representation of a `Rare` respecting ***strictly*** the following format :
  ```
  Timbre de code <code> datant de <annee> (provenance <pays>)
   ayant pour valeur faciale <valeur faciale> francs
  ```
  in only one line then an End Of Line followed by
  ```
  Nombre d'exemplaires -> <exemplaires>
  ```
  where `<code>` is to be replaced by the code of the stamp, `<annee>` by its issue year, `<pays>` by its country of origin and `<valeur faciale>`, by its denomination value and `<exemplaires>` by the number of inventoried copies ;

Commemorative stamps The class `Commemoratif` has to provide as public methods :

- constructors that conform to the provided `main()`. Its parameters, if they are not provided, they have the default values that are indicated for the basic stamps ;

- a method `toString()` producing a representation of a `Commemoratif` respecting ***strictly*** the following format :
  ```
  Timbre de code <code> datant de <annee> (provenance <pays>)
   ayant pour valeur faciale <valeur faciale> francs
  ```
  in only one line then an End Of Line followed by
  ```
  Timbre celebrant un evenement
  ```
  where `<code>` is to be replaced by the code of the stamp, `<annee>` by its issue year, `<pays>` by its country of origin and `<valeur faciale>`, by its denomination value.

It should be possible to calculate the selling price of a stamp `Commemoratif` using a method `double vente()`.

You are therefore asked to code the hierarchy of classes according to this description. Avoid code duplication, pay attention to the access modifiers of the attributes and name the classes as suggested in the statement.

By the way, you will assume that the classes `Rare` and `Commemoratif` do not inherit the one from the other.

1.2 Execution example

```
Timbre de code Guarana-4574 datant de  1960 (provenance Mexique) ayant pour valeur faciale 0.2 francs
Nombre d'exemplaires -> 98
Prix vente : 3360.0 francs

Timbre de code 700eme-501 datant de  2002 (provenance Suisse) ayant pour valeur faciale 1.5 francs
Timbre celebrant un evenement
Prix vente : 105.0 francs

Timbre de code Setchuan-302 datant de  2004 (provenance Chine) ayant pour valeur faciale 0.2 francs
Prix vente : 6.000000000000001 francs

Timbre de code Yoddle-201 datant de  1916 (provenance Suisse) ayant pour valeur faciale 0.8 francs
Nombre d'exemplaires -> 3
Prix vente : 6000.0 francs
```

2 Exercise 2 — Construction Kit

The purpose of this exercise is to model in a basic way construction games, of Lego ® type.

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] https://d396qusza40orc.cloudfront.net/intropoojava/assignments-data/Lego.java

2. save the downloaded file as `Lego.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 `Lego.java`) in "OUTPUT submission" (not in "Additional"!).

The provided creates a construction (using simple or composite pieces) and displays it.

A possible execution example is provided further below.

2.2 Classes to produce

A construction consists of a list of components ("composants" in French). It is characterized by the maximal number of components that can fit in it.

A component (`Composant`) is made of :

- a piece ;
- a quantity (number of times that the piece is used) ;

A piece is characterized by its name. It can be *simple* or *composée* (composite).

A simple piece is characterized by its orientation, a string that can be « *gauche* » (left), « *droite* » (right) or « *aucune* » (none).

A composite piece is characterized by the *list* of pieces that compose it **(they are not necessarily simple)** as well as the maximal number of pieces that can fit in it.

You are asked to code the classes : `Piece`, `Composant`, `Simple` (for the simple pieces), `Composee` (for the composite pieces) and `Construction` that correspond to the above description.

All the lists will be modeled as `ArrayList`. The insertions in these lists will be always done **at the end of the list**.

The `Piece` class The public methods of the class `Piece` will be :

- a constructor that allows to initialize the name of a piece using a value passed as parameter ;
- a getter for this attribute, `getNom()` ;
- a method `toString()` producing a representation of a `Piece` respecting ***strictly*** the following format : `<nom>` (name of the piece).

The `Composant` class The public methods of the class `Composant` will be :

- a constructor that allows to initialize the piece of the `Composant` and its quantity using values passed as parameters (the piece of the composite will be initialized using a reference to a piece passed as parameter) ;
- the getters `getPiece()` and `getQuantite()`.

The `Simple` class The public methods of the class `Simple` will be :

- a constructor that conforms to the provided `main()` method and that allows to initialize the name of the simple piece and its orientation using values passed as parameters. If no value is given for the orientation, the value by default `"aucune"` (none) will be used ;
- a getter `getOrientation()` ;
- a method `toString()` producing a representation of a `Simple` piece respecting ***strictly*** the following format : `<nom> [<orientation>]` where `<nom>` is the name of the piece and `<orientation>` its orientation. The orientation will only be present if its value is different from `"aucune"`.

The `Composee` class The public methods of the class `Composee` will be :

- a constructor that conforms to the provided `main()` method and that allows to initialize the name of the composite piece and the maximal number of pieces that can fit in it ;
- a method `int taille()` returning the number of pieces that compose the composite piece ;
- a method `int tailleMax()` returning the maximal number of pieces that can fit in the composite piece ;
- a method `void construire()` that conforms to the provided `main()` method, allowing to add a piece in the list of pieces that compose the composite piece. This insertion will only be possible if the maximal number of pieces is not reached. If there is an insertion attempt even though the list is full, the message
  `Construction impossible`
  will be displayed (followed by an End Of Line),
- a method `toString()` producing a representation of a `Composee` piece respecting ***strictly*** the following format :
  `<nom>` (<description piece1>, <description piece2>, ... <description pieceN>
)
  (without End Of Line) where `<nom>` is the name of the piece and `<descripion pieceX>` is the representation in the format of a `String` of the Xth piece composing the composit piece. All the pieces that compose the composite piece should be present (see the execution example further below).

The `Construction` class The public methods of the class `Construction` will be :

- a constructor that conforms to the provided `main()` method and allowing to initialize the maximal number of components that can fit in the construction ;
- a method `int taille()` returning the number of components that compose the construction ;
- a method `int tailleMax()` returning the maximal number of components that can fit in the construction ;

- a method `void ajouterComposant` that conforms to the provided `main()` method, allowing to insert a component in the list of components which compose the construction. This insertion will only be possible if the maximal number of components is not reached. If there is an insertion attempt even though the list is full, the message
  `Ajout de composant impossible`
  will be displayed (followed by an End Of Line) ; **the component to be inserted will be created from the arguments provided to the method `ajouterComposant()`** ;

- a method `toString()` producing a representation of a `construction` respecting ***strictly*** the following format :

  ```
  <description piece1> (quantite <quantite1>)
  ...

  <description pieceN> (quantite <quantiteN>)
  ```

  it consists of a representation of all the components of the construction (separated by an End Of Line) ; `<descripion pieceX>` is the representation in the format of a `String` of the piece of the xth component of the construction and `<quantiteX>` the quantity of that piece (see the execution example further below).

You are therefore asked to code the hierarchy of classes according to this description. Avoid code duplication, pay attention to the access modifiers of the attributes and name the classes as suggested in the statement.

A possible execution example is provided further below for a construction consisting of 59 basic bricks, a door and two windows.

2.3 Execution example

```
Affichage du jeu de construction :
brique standard (quantite  59)
porte (cadran porte gauche,battant gauche) (quantite 1)
fenetre (cadran fenetre,volet gauche,volet droit) (quantite 2)
```