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

Troisième devoir (noté)
Héritage

J. Sam & J.-C. Chappelier

Ce devoir comprend deux exercices à rendre.

1 Exercice 1 — Philatélie

Un philatéliste souhaite estimer à quel prix il pourrait vendre ses timbres. Le but de cet exercice est d'écrire un programme lui permettant de le faire.

1.1 Description

Télécharger le programme fourni sur le site du cours [1] et le compléter.

ATTENTION : vous ne devez modifier ni le début ni la fin du programme, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc primordial de respecter la procédure suivante (les points 1 et 3 concernent spécifiquement les utilisateurs d'Eclipse) :

1. désactiver le formatage automatique dans Eclipse :
   `Window > Preferences > Java > Editor > Save Actions`
   (et décocher l'option de reformatage si elle est cochée)
2. sauvegarder le fichier téléchargé sous le nom `Philatelie.java` (avec une majuscule, notamment). Si vous travaillez avec Eclipse vous ferez cette sauvegarde à l'emplacement `[dossierDuProjetPourCetExercice]/src/`;
3. rafraîchir le projet Eclipse où est stocké le fichier (clic droit sur le projet > refresh) pour qu'il le prenne en compte ;

1. `https://d396qusza40orc.cloudfront.net/intropoojava/assignments-data/Philatelie.java`

4. écrire le code à fournir entre ces deux commentaires :

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

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

5. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs données plus bas ;
6. rendre le fichier modifié (toujours `Philatelie.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

Le code fourni :

— crée une collection de timbres (rares ou commémoratifs) ;
— et affiche le prix de chaque timbre de cette collection.

La hiérarchie de `Timbre` manque et c'est ce qu'il vous est demandé d'écrire.

Un timbre est caractérisé par : son *code*, son *année d'émission*, son *pays d'origine* et sa *valeur faciale* en francs (l'équivalent en francs de la valeur apposée sur le timbre).

Notre philatéliste distingue deux grandes catégories de timbres, se distinguant notamment par les modalités du calcul du prix de vente :

— les timbres *rares* (classe `Rare`) : dotés d'un attribut supplémentaire indiquant le nombre d'exemplaires recensés dans le monde ;

— les timbres *commémoratifs* (`Commemoratif`) : sans attribut spécifique particulier.

Prix de vente des timbres Le prix de vente d'un *timbre* quelconque est sa valeur faciale si le timbre a moins que cinq ans. Sinon le prix de vente vaut la valeur faciale multipliée par l'âge du timbre et par le coefficient 2.5.

Le prix de vente d'un timbre *rare* part du calcul d'un *prix de base* : de 600 francs si le nombre d'exemplaires recensés est inférieur à 100, de 400 francs si le nombre d'exemplaires est entre 100 et 1000 et 50 francs sinon. Le prix de vente du timbre est alors donné par la formule `prix_base * (age_timbre / 10.0)`. **Les différentes constantes impliquées dans le calcul sont données dans le fichier fourni**.

Le prix de vente d'un timbre *commémoratif* est le double du prix de vente d'un timbre quelconque.

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

Timbres de base Les méthodes publiques de la classe `Timbre` sont :

- des constructeurs conformes à la méthode `main()` fournie, avec l'ordre suivant pour les paramètres : le code, l'année d'émission, le pays d'origine, et la valeur faciale ; chacun de ces paramètres admet une valeur par défaut : `VALEUR_TIMBRE_DEFAUT` pour la valeur faciale, `PAYS_DEFAUT` pour le pays, `ANNEE_COURANTE` pour l'année et `CODE_DEFAUT` pour le code ; la construction d'un timbre peut donc se faire avec zéro, un, deux, trois ou quatre paramètres (en respectant l'ordre fourni plus haut) ;
- une méthode `vente()` retournant le prix de vente sous la forme d'un `double` ;
- une méthode `toString()` produisant une représentation d'un `Timbre` respectant ***strictement*** le format suivant :
  `Timbre de code <code> datant de <annee> (provenance <pays>) ayant pour valeur faciale <valeur faciale> francs`
  en une seule ligne où `<code>` est à remplacer par le code du timbre, `<annee>` par son année d'émission, `<pays>`, par son pays d'origine et `<valeur faciale>`, par sa valeur faciale ;
- la méthode `age()` retournant l'âge du timbre sous la forme d'un `int` (différence entre `ANNEE_COURANTE` et l'année d'émission du timbre) ;
- les getters `getCode()`, `getAnnee()`, `getPays()` et `getValeurFaciale()`.

Timbres rares Comme méthodes publiques, la classe `Rare` doit en tout cas fournir :

— des constructeurs conformes à la méthode `main()` fournie, avec l'ordre suivant pour les paramètres : le code, l'année d'émission, le pays d'origine, la valeur faciale et le nombre d'exemplaires recensés. Le nombre d'exemplaires doit toujours être fourni lors des appels aux constructeurs. Les autres paramètres, s'ils ne sont pas fournis, ont les valeurs par défaut indiquées pour les timbres de base ;

— un getter `getExemplaires()` ;

— une méthode `toString()` produisant une représentation d'un `Rare` respectant ***strictement*** le format suivant :
`Timbre de code <code> datant de <annee> (provenance <pays>) ayant pour valeur faciale <valeur faciale> francs`
en une seule ligne puis un saut de ligne suivi de
`Nombre d'exemplaires -> <exemplaires>`
où `<code>` est à remplacer par le code du timbre, `<annee>` par son année d'émission, `<pays>` par son pays d'origine, et `<valeur faciale>` par sa valeur faciale et `<exemplaires>` par le nombre d'exemplaires recensés.

Il doit être possible de calculer le prix de vente d'un timbre `Rare` au moyen d'une méthode `double vente()`.

Timbres commémoratifs Comme méthodes publiques, la classe `Commemoratif` doit en tout cas fournir :

— des constructeurs conformes à la méthode `main()` fournie. Ces paramètres, s'ils ne sont pas fournis, ont les valeurs par défaut indiquées pour les timbres de base ;

— une méthode `toString()` produisant une représentation d'un `Commemoratif` respectant ***strictement*** le format suivant :
`Timbre de code <code> datant de <annee> (provenance <pays>) ayant pour valeur faciale <valeur faciale> francs`
en une seule ligne puis un saut de ligne suivi de
`Timbre celebrant un evenement`
où `<code>` est à remplacer par le code du timbre, `<annee>` par son année d'émission, `<pays>` par son pays d'origine, et `<valeur faciale>` par sa valeur faciale.

Il doit être possible de calculer le prix de vente d'un timbre `Commemoratif` au moyen d'une méthode `double vente()`.

Il vous est donc demandé de coder la hiérarchie de classes découlant de cette description. vous éviterez la duplication de code, le masquage d'attributs et nommerez les classes tels qu'il vous est suggéré de le faire dans l'énoncé.

Par ailleurs vous considérerez que les classes `Rare` et `Commemoratif` n'héritent pas l'une de l'autre.

1.2 Exemple de déroulement

```
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 Exercice 2 — Kit de construction

Le but de cet exercice est de modéliser de façon basique des jeux de construction, de type Lego®.

2.1 Description

Télécharger le programme fourni sur le site du cours [2] et le compléter.

ATTENTION : vous ne devez modifier ni le début ni la fin du programme, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc primordial de respecter la procédure suivante (les points 1 et 3 concernent spécifiquement les utilisateurs d'Eclipse) :

1. désactiver le formatage automatique dans Eclipse :

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

`Window > Preferences > Java > Editor > Save Actions`
(et décocher l'option de reformatage si elle est cochée)

2. sauvegarder le fichier téléchargé sous le nom `Lego.java` (avec une majuscule, notamment). Si vous travaillez avec Eclipse vous ferez cette sauvegarde à l'emplacement `[dossierDuProjetPourCetExercice]/src/` ;
3. rafraîchir le projet Eclipse où est stocké le fichier (clic droit sur le projet > refresh) pour qu'il le prenne en compte ;
4. écrire le code à fournir entre ces deux commentaires :

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

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

5. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs données plus bas ;
6. rendre le fichier modifié (toujours `Lego.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

Le code fourni crée une construction (au moyen de pièces simples ou composées) et l'affiche.

Un exemple de déroulement possible est fourni plus bas.

2.2 Classes à produire

Une construction est constituée d'une liste de composants. Elle est caractérisée par le nombre maximal de composants pouvant la constituer.

Un composant est constitué :

— d'une pièce ;
— d'une quantité (nombre de fois qu'est utilisée la pièce) ;

Une pièce est caractérisée par son nom. Elle peut être *simple* ou *composée*.

Une pièce simple est caractérisée par son orientation, une chaîne de caractères qui peut être « *gauche* », « *droite* » ou « *aucune* ».

Une pièce composée est caractérisée par la *liste* des *pièces* la constituant **(pas forcément toutes simples)** ainsi que le nombre maximal de pièces qui peuvent la constituer.

Il vous est demandé de coder les classes : `Piece`, `Composant`, `Simple` (pour les pièces simples), `Composee` (pour les pièces composées) et `Construction` correspondant à la description précédente.

Toutes les listes seront modélisées au moyen de `ArrayList`. Les ajouts dans ces listes se feront toujours **en fin de liste**.

La classe `Piece` Les méthodes publiques de la classe `Piece` seront :

- un constructeur permettant d'initialiser le nom de la pièce au moyen d'une valeur passée en paramètre ;
- un getter `getNom()` ;
- une méthode `toString()` produisant une représentation d'une `Piece` respectant ***strictement*** le format suivant : `<nom>` (nom de la pièce).

La classe `Composant` Les méthodes publiques de la classe `Composant` seront :

- un constructeur permettant d'initialiser la pièce du composant et sa quantité au moyen de valeurs passées en paramètres (la pièce du composant sera initialisée au moyen d'une référence sur une pièce passée en paramètre) ;
- les getters `getPiece()` et `getQuantite()`.

La classe `Simple` Les méthodes publiques de la classe `Simple` seront :

- un constructeur conforme à la méthode `main()` fournie et permettant d'initialiser le nom de la pièce simple et son orientation au moyen de valeurs passées en paramètres. Si aucune valeur n'est donnée pour l'orientation, la valeur par défaut `"aucune"` sera utilisée ;
- un getter `getOrientation()` ;
- une méthode `toString()` produisant une représentation d'une pièce `Simple` respectant ***strictement*** le format suivant :
  `<nom> [<orientation>]`
  où `<nom>` est le nom de la pièce est `<orientation>` son orientation. L'orientation ne sera présente que si sa valeur est différente de `"aucune"`.

La classe `Composee` Les méthodes publiques de la classe `Composee` seront :

- un constructeur conforme à la méthode `main()` fournie et permettant d'initialiser le nom de la pièce composée et le nombre maximal de pièces

pouvant la constituer ;

- une méthode `int taille()` retournant le nombre de pièces constituant la pièce composée ;
- une méthode `int tailleMax()` retournant le nombre maximal de pièces pouvant constituer la pièce composée ;
- une méthode `void construire()` conforme à la méthode `main()` fournie, permettant d'ajouter une pièce dans la liste des pièces constituant la pièce composée. Cet ajout ne sera possible que si le nombre de pièces maximal n'est pas atteint. S'il y a tentative d'ajout alors que la liste est pleine, le message
  `Construction impossible`
  sera affiché (suivi d'un saut de ligne),
- une méthode `toString()` produisant une représentation d'une pièce `Composee` respectant ***strictement*** le format suivant :
  `<nom>` (<description piece1>, <description piece2>, ... <description pieceN>)
  (sans saut de ligne) où `<nom>` est le nom de la pièce et `<descripion pieceX>` est la représentation sous forme d'une `String` de la Xieme pièce constituant la pièce composée. Toutes les pièces constituant la pièce composée devront être présentes (voir l'exemple de déroulement plus bas).

La classe `Construction` Les méthodes publiques de la classe `Construction` seront :

- un constructeur conforme à la méthode `main()` fournie et permettant d'initialiser le nombre maximal de composants pouvant constituer la construction ;
- une méthode `int taille()` retournant le nombre de composants constituant la construction ;
- une méthode `int tailleMax()` retournant le nombre maximal de composants pouvant constituer la construction ;
- une méthode `void ajouterComposant` conforme à la méthode `main()` fournie, permettant d'ajouter un composant dans la liste des composants constituant la construction. Cet ajout ne sera possible que si le nombre de composants maximal n'est pas atteint. S'il y a tentative d'ajout alors que la liste est pleine, le message
  `Ajout de composant impossible`

sera affiché (suivi d'un saut de ligne) ; **le composant à ajouter sera créé à partir des arguments fournis à la méthode `ajouterComposant()`** ;

— une méthode `toString()` produisant une représentation d'une `construction` respectant ***strictement*** le format suivant :

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

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

il s'agit d'une représentation de tous les composants de la construction (séparés par un saut de ligne) ; `<descripion pieceX>` est la représentation sous forme d'une `String` de la pièce du xième composant de la construction et `<quantiteX>` la quantité de cette pièce (voir l'exemple de déroulement plus bas).

Il vous est donc demandé de coder la hiérarchie de classes découlant de cette description. vous éviterez la duplication de code, le masquage d'attributs et nommerez les classes tels qu'il vous est suggéré de le faire dans l'énoncé.

Un exemple de déroulement possible est fourni plus bas pour une construction constituée de 59 briques de base, une porte et deux fenêtres.

2.3 Exemple de déroulement

```
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)
```