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

Quatrième devoir (noté) : Polymorphisme

J. Sam & J.-C. Chappelier

du 19 novembre au 07 décembre 2015

Ce devoir comprend deux exercices à rendre.

1 Exercice 1 — Agence de voyage

Un voyagiste souhaite que vous l'aidiez à gérer ses offres de voyage.

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 `Voyage.java` (avec une majuscule, notamment). Si vous travaillez avec Eclipse vous ferez cette sauvegarde à l'emplacement `[dossierDuProjetPourCetExercice]/src/`;

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

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 `Voyage.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

1.2 Le code à produire

Les options de voyages Notre voyagiste vend des kits de voyage composés de différentes *options*.

Il s'agit d'abord d'implémenter une classe `OptionVoyage` permettant de représenter de telles options.

Une *option* (classe `OptionVoyage`) est caractérisée par :

— son *nom*, une chaîne de caractères ;

— et son *prix forfaitaire* (un `double`).

La classe `OptionVoyage` comportera :

— un constructeur initialisant les attributs au moyen de valeurs passées en paramètre et dans un ordre compatible avec le `main` fourni ;

— une méthode `getNom` retournant le nom de l'option ;
— une méthode `double prix()` retournant le prix forfaitaire de l'option ;
— une méthode `toString` produisant une représentation de l'option sous la forme d'une chaîne de caractères, selon le format suivant :

```
<nom> -> <prix> CHF
```

où <nom> est le nom de l'option et <prix> est son prix.

Il vous est demandé d'implémenter la classe `OptionVoyage` en respectant une bonne encapsulation.

Cette partie de votre programme peut-être testée au moyen de la portion de code comprise entre `// TEST 1` et `// FIN TEST 1`.

Les options de voyage peuvent bien sûr se décliner en différentes sous-classes. Il s'agit ici d'en modéliser deux : les moyens de transport (classe `Transport`) et le logement pendant le voyage (classe `Sejour`).

La classe `Sejour` Une instance de `Sejour` sera caractérisée par *le nombre de nuits* (un entier) et le *prix par nuit* (un double).

Le prix d'un séjour est simplement le nombre de nuits multiplié par le prix par nuit, auquel on ajoutera le prix forfaitaire de l'option.

La classe `Transport` Une instance de `Transport` sera caractérisée par un booléen indiquant si le trajet est long.

Le prix du transport vaut la constante `TARIF_LONG` (`1500.0`) si le trajet est long et `TARIF_BASE` (`200.0`) sinon, auquel on ajoutera le prix forfaitaire de l'option. **Les constantes seront publiquement accessibles.**

Faites maintenant en sorte que la classe `OptionVoyage` se spécialise en deux sous-classes : `Transport` et `Sejour` répondant à la description précédente.

La hiérarchie de classes sera dotée :

— de constructeurs conformes au `main` fourni. Les arguments sont dans l'ordre : le nom, le prix forfaitaire et un booléen (valant true si le trajet est long et false sinon) pour les `Transport`. Les arguments pour le constructeur de `Sejour` sont dans l'ordre : le nom, le prix forfaitaire, le nombre de nuits et le prix par nuit. **Par défaut, un `Transport` a un trajet court**.

— de redéfinitions spécifiques de la méthode `prix`. Ces spécialisations ne contiendront aucune
duplication de code et seront utilisables de façon polymorphique.

Cette partie de votre programme peut être testée au moyen de la portion de code comprise entre `// TEST 2` et `// FIN TEST 2`.

Kit de voyage Le voyagiste vend des kits composés de plusieurs options.

Il vous est demandé de coder une classe `KitVoyage` comme une «collection hétérogène» de `OptionVoyage` (un `ArrayList`).

La classe `KitVoyage` sera également caractérisée par le *départ* et la *destination* du kit (deux `String`).

La classe `KitVoyage` sera dotée :

- d'un constructeur compatible avec le `main` fourni (voir la portion de code entre `// TEST 3` et `// FIN TEST 3`);
- d'une méthode `double prix()` qui calculera le prix du kit comme la somme du prix de toutes ses options ;
- d'une méthode `toString`, générant une représentation du kit sous la forme d'une `String`, selon le format suivant :
  ```
  Voyage de <depart> à <destination>, avec pour options :
  - <nom option1> -> <prix option1> CHF
  - ....
  - <nom optionN> -> <prix optionN> CHF
  Prix total : <prix du kit> CHF
  ```
  où `<depart>` est le départ du kit, `<destination>` sa destination et `<prix du kit>` son prix. La chaîne construite se terminera par `\n`.
- d'une méthode `ajouterOption`, compatible avec le `main` fourni et permettant d'ajouter une `OptionVoyage` à la collection d'options du kit (les options seront ajoutées en fin de collection). Si l'argument de `ajouterOption` vaut `null`, il ne sera pas ajouté à la collection.
- une méthode `annuler` vidant la collection d'options (utiliser la méthode `clear` des `ArrayList`) ;
- une méthode `getNbOptions` retournant le nombre d'options de voyage du kit.

Cette partie de votre programme peut être testée au moyen de la portion de code comprise entre `// TEST 3` et `// FIN TEST 3`.

1.3 Exemple de déroulement

```
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
```

2 Exercice 2 — Bataille navale

Nous nous intéressons dans cet exercice à modéliser de façon très basique un jeu de bataille navale avec des *navires*. Ces derniers pourront être des *navires pirates* et des *navires marchands*.

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

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

2. sauvegarder le fichier téléchargé sous le nom `Piraterie.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 `Piraterie.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

Il vous est demandé de compléter le code selon la description en quatre parties qui suit.

2.2 Le code à produire

Il vous est demandé de compléter le code selon la description qui suit.

La classe `Navire` un `Navire` est caractérisé par :

— une *coordonnée en x* et une *coordonnée en y* permettant de repérer sa position dans l'espace (deux entiers pour simplifier) ;
— un *drapeau* (sous la forme d'un entier) indiquant à quel camp appartient le navire ;
— une information indiquant s'il est *détruit* (coulé).

La classe `Navire` comportera :

— un constructeur, conforme avec la méthode `main` fournie, initialisant les coordonnées et le drapeau du navire au moyen de valeurs passées en paramètre ; le navire « construit » ne sera pas marqué comme *détruit* d'emblée (!) ;

— les « getters » `getX()`, `getY()` et `getDrapeau()` retournant les coordonnées du navire et son drapeau ;
— la méthode `boolean estDetruit()` retournant vrai si le navire est coulé ;
— la méthode `distance`, conforme à la méthode `main` fournie, et retournant la distance (un `double`) séparant le navire d'un autre navire ; la distance entre deux points (x_1, y_1) et (x_2, y_2) se calcule selon la formule $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$;
— la méthode `void avance(int unitsX, int unitsY)` permettant de déplacer le navire de `unitsX` unités horizontalement et de `unitsY` unités verticalement (`unitsX` et `unitsY` peuvent être négatifs) ;
— la méthode `void couler()` permettant de marquer le navire comme détruit (coulé).

Il vous est demandé d'implémenter la classe `Navire`, de manière à ce que les contraintes suivantes soient respectées :

1. les coordonnées du navire ne peuvent être inférieures à 0 ou supérieures aux constantes fournies `Piraterie.MAX_X` et `Piraterie.MAX_Y` ; les coordonnées que l'on tente donc d'affecter à la construction ou via d'autre méthodes devront être plafonnées à ces valeurs ;
2. La classe doit être bien encapsulée.

Les sous-classes `Pirate` et `Marchand` Vous ferez ensuite en sorte que la classe `Navire` se spécialise en deux sous-classes : les navires pirates (classe `Pirate`) et les navires marchands (classe `Marchand`).

Les navires pirates peuvent être endommagés avant d'être détruits (un booléen servira à indiquer s'ils sont endommagés). L'initialisation de cet attribut se fera au moyen d'une valeur passée en paramètre.

La hiérarchie de classes sera dotée :

— de constructeurs conformes à la méthode `main` fournie ;
— d'une méthode polymorphique `String getNom()` retournant le nom générique du navire : `"Bateau"` dans le cas de la super-classe, `"Bateau pirate"` pour les navires pirates et `"Bateau marchand"` pour les navires marchands ;

— de la redéfinition de la méthode `toString` générant la représentation sous la forme d'une `String` qui respectera ***strictement*** le format suivant : `<nom générique> avec drapeau <drapeau> en (<x>,<y>) -> <etat>` où `<nom générique>` est le nom générique du navire, `<drapeau>`, la valeur de son drapeau, `<x>` et `<y>`, ses coordonnées en x et y et `<etat>` l'une des chaînes : `"intact"`, `"ayant subi des dommages"` ou `"détruit"` selon que le bateau est intact, a subi des dommages mais n'est pas encore coulé ou est détruit (coulé) ;
— la classe `Pirate` contiendra également une méthode `estEndommage()` retournant `true` si le bateau pirate est endommagé et `false` dans le cas contraire.

Cette partie de votre programme peut être testée par la portion de la méthode `main` fournie comprise entre `// Test de la partie 1` et `// Test de la partie 2` (non comprise, voir le code fourni).

Mauvaises rencontres Pour simuler le jeu de bataille navale, on adopte le critère suivant :

si deux navires sont ennemis (ont des drapeaux différents) et que la distance les séparant est inférieure à `Piraterie.RAYON_RENCONTRE` ils se confrontent, sinon rien ne se passe.

Il vous est donc demandé de doter la hiérarchie d'une méthode `rencontre` qui va :
— tester si un navire est suffisamment proche d'un autre ;
— s'ils ont des drapeaux différents ;
— et dans ce cas faire se confronter les deux navires (méthode `combat`).

Les règles de confrontation sont les suivantes :

1. si un navire pirate `b1` se confronte à un autre navire `b2` : `b2` reçoit un boulet (méthode `recoitBoulet`) ; `b1` reçoit aussi un boulet si `b2` est un navire pirate ;
2. dans tous les autres cas rien ne se passe ;
3. un navire marchand qui reçoit un boulet est coulé (détruit) ;
4. un navire pirate non endommagé qui reçoit un boulet est endommagé ;
5. un navire pirate endommagé qui reçoit un boulet est coulé.

Il vous est demandé de programmer les méthodes suggérées en respectant les contraintes suivantes :

1. la hiérarchie de classes comportera une méthode polymorphique `boolean estPacifique()` retournant « vrai » pour navire marchand et « faux » sinon ; cette méthode sera utilisée pour mettre en oeuvre les traitements décrits ;
2. la méthode `rencontre` sera conforme à la méthode `main` fournie ;
3. vous considérerez que les méthodes `combat` et `recoitBoulet` ne peuvent être définies concrètement pour un navire quelconque.

Cette partie de votre programme peut être testée par la portion de la méthode `main` fournie après `// Test de la partie 2` (voir le code fourni).

2.3 Exemple de déroulement

```
***Test de la partie 1***

Bateau pirate avec drapeau 1 en (0,0) -> ayant subi des dommages
Bateau marchand avec drapeau 2 en (25,0) -> intact
Distance: 25.0
Quelques déplacements horizontaux et verticaux
Bateau pirate avec drapeau 1 en (75,100) -> ayant subi des dommages
Bateau marchand avec drapeau 2 en (25,0) -> intact
Un déplacement en bas:
Bateau pirate avec drapeau 1 en (75,95) -> ayant subi des dommages
Après destruction:
Bateau pirate avec drapeau 1 en (75,95) -> détruit
Bateau marchand avec drapeau 2 en (25,0) -> détruit

***Test de la partie 2***

Bateau pirate et marchand ennemis (trop loin):
Bateau pirate avec drapeau 1 en (0,0) -> intact
Bateau marchand avec drapeau 2 en (0,25) -> intact
Après la rencontre:
Bateau pirate avec drapeau 1 en (0,0) -> intact
Bateau marchand avec drapeau 2 en (0,25) -> intact

Bateau pirate et marchand ennemis (proches):
Bateau pirate avec drapeau 1 en (0,0) -> intact
Bateau marchand avec drapeau 2 en (2,0) -> intact
Après la rencontre:
Bateau pirate avec drapeau 1 en (0,0) -> intact
Bateau marchand avec drapeau 2 en (2,0) -> détruit

Bateau pirate et marchand amis (proches):
Bateau pirate avec drapeau 1 en (0,0) -> intact
Bateau marchand avec drapeau 1 en (2,0) -> intact
```

```
Après la rencontre:
Bateau pirate avec drapeau 1 en (0,0) -> intact
Bateau marchand avec drapeau 1 en (2,0) -> intact

Deux bateaux pirates ennemis intacts (proches):
Bateau pirate avec drapeau 1 en (0,0) -> intact
Bateau pirate avec drapeau 2 en (2,0) -> intact
Après la rencontre:
Bateau pirate avec drapeau 1 en (0,0) -> ayant subi des dommages
Bateau pirate avec drapeau 2 en (2,0) -> ayant subi des dommages

Un bateau pirate intact et un avec dommages, ennemis:
Bateau pirate avec drapeau 1 en (0,0) -> ayant subi des dommages
Bateau pirate avec drapeau 3 en (0,2) -> intact
Après la rencontre:
Bateau pirate avec drapeau 1 en (0,0) -> détruit
Bateau pirate avec drapeau 3 en (0,2) -> ayant subi des dommages

Deux bateaux pirates ennemis avec dommages:
Bateau pirate avec drapeau 2 en (2,0) -> ayant subi des dommages
Bateau pirate avec drapeau 3 en (0,2) -> ayant subi des dommages
Après la rencontre:
Bateau pirate avec drapeau 2 en (2,0) -> détruit
Bateau pirate avec drapeau 3 en (0,2) -> détruit
```