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

# Quatrième devoir (noté) : Polymorphisme

J. Sam & J.-C. Chappelier

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 [1] le programme fourni sur le site du cours 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/`;
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 :

1. Nous parlons bien ici de « *télécharger* », i.e. sauvegarder le fichier tel quel, et non pas de le copier-coller depuis le navigateur.

```
/*****************************************
 * 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
```

cd

# 2 Exercice 2 — Rallyes

Un organisateur de rallyes auto-moto vous demande de l'aide pour organiser ses courses.

## 2.1 Description

Télécharger[2] le programme fourni sur le site du cours 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 `Course.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 ;

2. Nous parlons bien ici de « *télécharger* », i.e. sauvegarder le fichier tel quel, et non pas de le copier-coller depuis le navigateur.

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

## 2.2 Le code à produire

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

**1) la classe `Vehicule`** Il s'agit d'abord d'implémenter une classe `Vehicule` permettant de représenter un véhicule participant aux courses.

Un *véhicule* est caractérisée par :

- son *nom*, une chaîne de caractères comme « Ferrari » par exemple ;
- sa *vitesse maximale* (un `double`) ;
- son *poids* en kg (un `int`) ;
- et le niveau de *carburant* de son réservoir (un entier).

La classe `Vehicule` comportera :

- un constructeur, conforme à la méthode `main` fournie, initialisant les attributs au moyen de valeurs passées en paramètre et un constructeur par défaut initialisant le nom à *"Anonyme"*, le niveau de carburant à zéro, la vitesse maximale à 130 et le poids à 1000 ;
- une méthode `toString` produisant une `String` contenant toutes les caractéristiques du véhicule sauf le niveau de carburant en respectant ***strictement*** le format suivant :

  `<nom> -> vitesse max = <vitesse max> km/h, poids = <poids> kg`

  où `<nom>` est le nom du véhicule, `<vitesse max>` sa vitesse maximale et `<poids>`, son poids ;

— une méthode `meilleur(Vehicule autreVehicule)` retournant `true` si l'instance courante a une strictement meilleure performance que `autreVehicule`;
— les « getters » `getNom()`, `getVitesseMax()`, `getPoids()` et `getCarburant()`.

Finalement, la classe devra comporter et utiliser une méthode utilitaire `double performance()`. Cette méthode devra retourner une estimation de la performance du véhicule comme le rapport entre sa vitesse maximale et son poids (plus le véhicule est léger et rapide, meilleure est sa performance car il consomme moins d'énergie) ;

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

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

**2) Voitures et motos** Les véhicules participant aux rallyes peuvent être soit des voitures soit des motos.

Une voiture (classe `Voiture`) est caractérisée par une information supplémentaire indiquant sa catégorie (« course » ou « tourisme »).

Une moto (classe `Moto`) est caractérisée par un booléen indiquant si elle possède un *Sidecar*.

Programmez maintenant la hiérarchie de classes vous permettant de représenter ces deux sortes de véhicules en la dotant :

— de constructeurs conformes au `main` fourni (dans la portion de code entre `// TEST 2` et `// FIN TEST 2`); **par défaut, une `Moto` n'a pas de Sidecar** ; il n'est pas nécessaire de tester la validité de la valeur des arguments dans les constructeurs ;
— de redéfinitions spécifiques de la méthode `toString` ; ces spécialisations ne contiendront pas de duplication de code.

Par ailleurs :

— la représentation d'une voiture sous la forme d'une `String` respectera ***strictement*** le format suivant :

```
<nom> -> vitesse max = <vitesse max> km/h, poids = <poids> kg, Voiture de <categorie>
```

où <nom> est le nom du véhicule, <vitesse max> sa vitesse maximale, <poids>, son poids et <categorie>, sa catégorie ;

— la représentation d'une moto sous la forme d'une `String` respectera ***strictement*** le format suivant :

```
<nom> -> vitesse max = <vitesse max> km/h, poids = <poids> kg, Moto, avec sidecar
```

si elle possède un « Sidecar », ou sinon :

```
<nom> -> vitesse max = <vitesse max> km/h, poids = <poids> kg, Moto
```

avec <nom> le nom du véhicule, <vitesse max> sa vitesse maximale, et <poids>, son poids.

Vous doterez enfin la classe `Voiture` d'un getter `getCategorie()`.

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

**3) les classes `GrandPrix` et `Rallye`** Il vous est demandé maintenant de coder une classe `GrandPrix` comme une « collection hétérogène » de véhicules. Cette collection représente l'ensemble des véhicules participant à une course. Elle sera modélisée au moyen d'un `ArrayList`.

Cette classe héritera d'une classe `Rallye` qui contient uniquement une méthode `boolean check()`. Cette méthode doit permettre de vérifier si les véhicules ont le droit de courir ensemble. La méthode `check` **ne peut être définie concrètement dans la classe `Rallye`**.

La classe `GrandPrix` sera dotée :

— d'une méthode `ajouter` permettant d'ajouter un véhicule à l'ensemble des participants (l'ajout se fera en fin de collection) ; cette méthode sera conforme à la méthode `main` fourni (dans la portion de code entre `// TEST 3` et `// FIN TEST 3`) ;

— pour un rallye de type `GrandPrix` les voitures n'ont pas le droit de courir avec les deux roues ; les motos ayant un «Sidecar» **ne sont pas considérées comme des véhicules à deux roues** ; les deux roues ont le droit de courir ensemble.

Pour tester la compatibilité des véhicules, vous doterez **la hiérarchie** de `Vehicule` d'une méthode :

```
boolean estDeuxRoues()
```

retournant `true` quand un véhicule est de type deux roues et `false` dans le cas contraire. Vous considérerez qu'**un véhicule quelconque n'est pas un deux roues**.

Cette partie de votre programme peut être testée par la portion de la méthode `main` fournie comprise entre `// TEST 3` et `// FIN TEST 3` (voir le code fourni).

**4) La course est lancée** Complétez la classe `GrandPrix` de sorte à lui ajouter une méthode `void run(int tours)` simulant le déroulement de la course selon l'algorithme suivant :

- commencer par tester si les véhicules ont le droit de courir ensemble ; le message *"Pas de Grand Prix"* suivi d'un saut de ligne sera affiché dans le cas contraire et la méthode `run` devra terminer son exécution ;
- quand la course a lieu : pour chaque véhicule, déduire autant de carburant que de `tours` ; seuls les véhicules à qui il reste du carburant ($> 0$) arrivent sur la ligne d'arrivée ;
- parmi les véhicules qui ont atteint la ligne d'arrivée, sélectionner le plus performant (celui qui est meilleur que tous les autres) et l'afficher en respectant ***strictement*** le format suivant :
  `Le gagnant du grand prix est :`
  `<representation>`
  où `<representation>` est la représentation du véhicule gagnant sous la forme d'une `String`, telle que produite par `toString`.
  Si aucun véhicule n'atteint la ligne d'arrivée, afficher le message
  `Elimination de tous les vehicules`
  suivi d'un saut de ligne.

Cette partie de votre programme peut être testée par la portion de la méthode `main` fournie comprise entre `// TEST 4` et `// FIN TEST 4` (voir le code fourni).

## 2.3 Exemple de déroulement

```
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
```