Programmation Orientée Objet
IG2I, LE2 – 2018-2019

TP2 – Courses de F1

Diego Cattaruzza, Philippe Kubiak, Maxime Ogier

Contexte et objectif du TP

Dans ce TP, nous poursuivons la modélisation et la gestion d'une écurie de Formule 1. Il s'agit de l'ensemble de l'équipe engagée dans le championnat du monde de Formule 1, qui englobe la partie technique et l'équipe sportive.

Nous allons donc créer des objets pour gérer une écurie de Formule 1. Pour qu'elle obtienne de bons résultats, il est très important d'assurer que seuls les pilotes puissent conduire les Formules 1, et seuls les techniciens puissent conduire les camions.

Ce TP permet ainsi d'illustrer les notions suivantes :

- l'héritage ;
- les classes abstraites ;
- les tableaux.

Bonnes pratiques à adopter

Pour réaliser ce TP, il vous est très fortement recommandé d'utiliser la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez. Par ailleurs, il est aussi important que votre propre code soit commenté au format Javadoc (commentaire commençant par "`/**`" au-dessus des définitions des attributs et méthodes). Ceci vous permet (1) de spécifier rigoureusement le comportement d'une méthode au moment où vous l'implémentez, (2) de savoir ce que fait une méthode sans avoir besoin de regarder son code (qui d'ailleurs peut ne pas être accessible), et donc (3) de faciliter le partage et la réutilisabilité du code.

1 Héritage

Nous abordons dans cette section une des notions fondamentales de la programmation orientée objet : l'héritage. La Section 4.1 présente quelques informations sur l'héritage en Java.

Jusqu'ici nous avons dit que dans notre écurie de Formule 1, nous considérions des personnes et des voitures sans donner plus de précisions. Mais en réalité, parmi les personnes, certaines sont des pilotes de Formule 1, et d'autres sont des techniciens. Pour les pilotes, il est nécessaire de définir le nombre de prix gagnés et le nombre d'abandons. Pour les techniciens, il est nécessaire de définir leur spécialité. De même, parmi les voitures, on peut distinguer les Formules 1 (caractérisées par un sponsor) et les camions servant à l'équipe technique (caractérisés par leur tonnage maximal).

Par ailleurs, le responsable des courses précise qu'un pilote doit pouvoir effectuer les opérations suivantes : gagner un prix, abandonner une course, se faire retirer un prix. Mais il n'est pas possible de gagner plusieurs courses en même temps ou de "désabandonner" une course. Et le responsable technique explique quant à lui que le tonnage des camions est une valeur non modifiable. Par contre les techniciens peuvent changer de spécialité.

Nous souhaitons enrichir notre modèle objet afin de prendre en compte ces spécialisations des personnes et des voitures.

Question 1. Modifiez le diagramme de classe afin d'y rajouter les spécialisation des personnes et des voitures.

Question 2. Écrivez une classe **Formule1** avec ses attributs. Ajoutez un constructeur par défaut et au moins 3 constructeurs par données. Ajoutez les accesseurs et mutateurs nécessaires. Ajoutez aussi une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur la Formule 1. Écrivez une méthode principale afin de réaliser les tests unitaires pour l'utilisation de cette classe.

Question 3. Écrivez une classe **Camion** avec ses attributs. Ajoutez un constructeur par défaut et au moins 3 constructeurs par données. Ajoutez les accesseurs et mutateurs nécessaires. Ajoutez aussi une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur le camion. Écrivez une méthode principale afin de de réaliser les tests unitaires pour l'utilisation de cette classe.

Question 4. Écrivez une classe **Pilote** avec ses attributs. Ajoutez un constructeur par défaut et au moins 2 constructeurs par données. Ajoutez les accesseurs et mutateurs nécessaires, ainsi que les méthodes de gestion des victoires et des abandons. Ajoutez aussi une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur le pilote. Écrivez une méthode principale

afin de réaliser les tests unitaires pour l'utilisation de cette classe.

Question 5. Écrivez une classe **Technicien** avec ses attributs. Ajoutez un constructeur par défaut et au moins 2 constructeurs par données. Ajoutez les accesseurs et mutateurs nécessaires. Ajoutez aussi une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur le technicien. Écrivez une méthode principale afin de de réaliser les tests unitaires pour l'utilisation de cette classe.

Pour la bonne réussite de toute l'équipe de Formule 1, seuls les pilotes doivent conduire les Formules 1 et seuls les techniciens doivent conduire les camions (même si on pourrait bien imaginer qu'un pilote puisse bien se débrouiller à la conduite d'un camion, on ne mettra pas un technicien à la conduite d'une Formule 1 si on veut maximiser nos chances de victoire).

Ainsi, il faut qu'on s'assure avant d'affecter une voiture à une personne que ces deux là soient bien compatibles. On pourrait envisager d'écrire dans la classe **Personne** une méthode **public boolean estCompatible(Voiture v)**. Mais on devrait alors tester toutes les combinaisons possibles de compatibilités selon les sous classes de **Personne** et **Voiture**. Pour le moment, il n'y en a pas beaucoup, mais si le code est amené à évoluer cette idée n'est pas bonne du tout. D'autant plus que si l'on rajoute une nouvelle sous-classe, il y a de forts risques pour qu'on ne se rappelle pas qu'il faut aussi modifier cette méthode **estCompatible()** ... On va donc commencer plus simplement, et utiliser le fait qu'on puisse redéfinir des méthodes dans les classes héritées. La méthode **estCompatible()** de la classe **Personne** renverra **false** (on suppose que par défaut une personne et une voiture ne sont pas compatibles) ; et dans chacune des sous-classes de **Personne**, on redéfinit la méthode **public boolean estCompatible(Voiture v)**. Cette fois-ci, connaissant le type de la sous-classe, on sait facilement s'il y a compatibilité ou non.

Question 6. Ajoutez dans la classe **Personne** une méthode **public boolean estCompatible(Voiture v)** qui retourne **false**. Ajoutez dans les classes **Pilote** et **Technicien** une méthode **public boolean estCompatible(Voiture v)** qui retourne **true** si le pilote/technicien est compatible avec la voiture en paramètre de la méthode. Vous pouvez utiliser l'opérateur **instanceof** qui compare un objet avec un type. Cet opérateur renvoie un booléen qui indique si l'objet est une instance d'une classe ou d'une sous-classe. Testez que les méthodes de compatibilité fonctionnent correctement.

Question 7. Modifiez la méthode d'affectation d'une voiture à une personne afin de ne réaliser cette affectation que s'il y a compatibilité entre la voiture et la personne. Testez afin de vérifier que tout fonctionne correctement.

2 Classes abstraites

Dans la Section précédente, nous avons défini la méthode **estCompatible(Voiture v)** dans la classe **Personne** et dans ses deux sous-classes. Supposons maintenant que pour se déplacer entre les différents grands-prix du championnat, notre équipe de Formule 1 achète des bus, embauche des chauffeurs et que seuls les chauffeurs puissent conduire les bus. On devrait alors écrire deux nouvelles classes **Bus** et **Chauffeur** et ajouter une méthode **estCompatible()** dans la classe **Chauffeur**. Êtes-vous certain de vous rappeler de faire cela ?

Par ailleurs, si l'on crée des objets de type **Personne** (qui ne sont donc ni des pilotes ni des techniciens), alors ils ne sont compatibles avec aucune voiture. On pourrait donc se poser la question suivante : aura-t-on besoin de définir des objets de type **Personne** qui ne soient pas d'un des sous-types de **Personne** ? Si la réponse est non, alors il faudrait empêcher qu'un utilisateur puisse définir des objets de type **Personne** et on forcerait ainsi à choisir explicitement un des sous-types de **Personne**.

Pour s'assurer de ne pas oublier de définir les méthodes de compatibilité à la création d'une sous classe de **Personne**, nous pouvons définir la méthode **estCompatible()** dans la classe **Personne** comme une méthode *abstraite*. Une méthode abstraite est une méthode dont on donne la signature, mais sans en décrire l'implémentation. Elle commence toujours par le mot clé **abstract**. Toutes les classes qui dérivent de la classe mère sont obligées d'implémenter les méthodes abstraites définies dans la classe mère. Ceci permet que lors de l'implémentation d'une nouvelle classe fille, le développeur sache quelles sont les méthodes obligatoires à implémenter. Par ailleurs, cette notion de méthode abstraite va de pair avec la notion de classe abstraite. En effet, une méthode abstraite ne peut être déclarée que dans une classe abstraite. Une classe abstraite est une classe qui peut contenir une ou plusieurs méthodes abstraites. Sa déclaration commence toujours par le mot clé **abstract**. Il n'est alors plus possible d'instancier une classe abstraite. Ceci permet donc de décrire des concepts incomplets, en factorisant des attributs ou comportements communs à un ensemble de sous-classes. Vous pouvez vous référer à la Section 4.2 pour plus d'informations sur les classes abstraites.

Question 8. Déclarez les classes **Personne** et **Voiture** comme étant des classes abstraites. Il n'est plus possible d'instancier des personnes ou des voitures (mais les pilotes et les techniciens sont toujours des personnes et les Formules 1 et les camions sont toujours des voitures).

Question 9. Déclarez dans la classe **Personne** la méthode **estCompatible(Voiture v)** comme une méthode abstraite. Testez vos modifications.

3 Association 1-N : création des écuries

Maintenant que tous les éléments sont prêts, nous allons pouvoir créer notre propre écurie de Formule 1. Pour participer au championnat, chaque écurie doit:

- avoir un nom ;
- avoir une équipe constituée d'au plus 5 personnes ;
- avoir une flotte d'au plus 5 véhicules.

Une écurie doit pouvoir :

- vérifier la présence d'une personne dans son équipe ;
- vérifier la présence d'une voiture dans sa flotte ;
- embaucher une personne ;
- acheter une voiture ;
- affecter une voiture à une personne ;
- restituer la voiture conduite par une personne.

Question 10. Modifiez le diagramme de classe afin d'y rajouter l'écurie.

Question 11. Créez une classe **Ecurie** avec ses attributs. Ajoutez un constructeur par donnée du nom de l'équipe ainsi que les accesseurs et mutateurs nécessaires et une méthode **toString()** qui renvoie une chaîne de caractères avec les informations sur l'équipe.

Question 12. Créez une méthode **public boolean estPresent(Personne p)** qui renvoie **true** si la personne **p** est déjà présente parmi l'équipe, **false** sinon. Notez que l'on peut vérifier la présence d'une personne dans l'équipe en comparant les identifiants (qui doivent être uniques). De la même manière, créez une méthode **public boolean estPresent(Voiture v)** qui renvoie **true** si la voiture **v** est déjà présente parmi la flotte de véhicules, **false** sinon.

Question 13. Créez une méthode **public boolean embaucher(Personne p)** qui permet d'embaucher une personne (qui peut être un pilote ou un technicien). On ne doit pas embaucher une personne déjà présente ni dépasser la taille maximale de l'équipe. Cette méthode renvoie un booléen qui indique si l'embauche a bien été réalisée ou non. De la même manière, créez une méthode **public boolean acheter(Voiture v)**. Testez ces méthodes.

Question 14. Créez une méthode **private Personne localiserPersonne(int id)** qui permet de localiser dans l'équipe la personne avec l'identifiant **id** si elle est présente. Dans ce cas, la méthode renvoie la personne en question. Si aucune personne avec l'identifiant **id** n'est présente dans l'équipe, alors la méthode renvoie **null**. De la même manière, créez une méthode **private Voiture localiserVoiture(int immat)**.

Question 15. Créez une méthode **public boolean affecter(int id, int immat)** qui affecte la voiture immatriculée **immat** à la personne identifiée par **id** si la voiture est dans la flotte et la personne dans l'équipe de l'écurie. Pensez à utiliser la méthode **affecter()** que vous avez défini dans la classe **Personne**. Testez cette méthode.

Question 16. Créez une méthode **public boolean restituer(int id)** qui restitue la voiture affectée à la personne définie par son identifiant **id**. Cette méthode renvoie un booléen qui indique si la restitution a bien eu lieu on non. Testez cette méthode.

4 Quelques notions utiles

4.1 Héritage

Une classe Java dérive toujours d'une autre classe, la classe **Object** quand rien n'est spécifié. Pour spécifier de quelle classe hérite une classe on utilise le mot-clé **extends** :

```java
public class D extends B {
   // La classe D hérite de la classe B
}
```

La classe **D** est dite *dérivée* et la classe **B** est dite *base*, *super-classe* ou *classe mère* de la classe **D**. On dit aussi que la classe **D** *dérive* de la classe **B** ou que **D** est une *sous-classe* de **B**. La classe **D** hérite de tous les attributs et méthodes de la classe **B**.

Chaque instance a deux références particulières :

- **this** : réfère l'instance elle-même ;
- **super** : réfère la partie héritée de l'instance.

Quand on construit une classe dérivée on doit s'assurer de construire la classe mère aussi. Par exemple :

```java
public class D extends B {
   private int attrD1;
   private int attrD2;

   // Constructeur par défaut
   public D () {
      // Appel du constructeur par défaut de la classe mère
      super();
   }

   // Constructeur par données 1
   public D (int attrD1, int attrD2) {
      // Appel du constructeur par défaut de la classe mère
      super();
      this.attrD1 = attrD1;
      this.attrD2 = attrD2;
   }

   // Constructeur par données 2
   public D (int attrD1, int attrD2, int attrB1) {
      // Appel du constructeur par données de la classe mère
      super(attrB1);
      this.attrD1 = attrD1;
      this.attrD2 = attrD2;
   }
}
```

Les méthodes de la classe dérivée peuvent surcharger et/ou redéfinir les méthodes de la classe mère.

- Redéfinition : même type de retour et mêmes paramètres.
- Surcharge : le type de retour peut être différent et les paramètres doivent être différents.

Par exemple, si la classe **B** définit les méthodes suivantes.

```java
public class B {
   // Méthodes
   public void methode1 () {
      System.out.println("Classe B !");
   }

   public int methode2 (int val) {
      return val + 1;
   }
}
```

Dans la classe **D**, nous pouvons par exemple avoir les méthodes suivantes :

```java
public class D extends B {
   // Méthodes
   // Rédéfinition de la méthode methode1
   public void methode1 () {
      System.out.println("Je suis de la classe D !");
   }

   // Surcharge de la méthode methode2
   public int methode2 (int val1, int val2) {
      return val1 + val2;
   }
}
```

Ainsi, il est possible d'écrire le code suivant :

```
B objB = new B();
D objD = new D();
objB.methode1(); // "Classe B !"
objD.methode1(); // "Je suis de la classe D !"
objB.methode2(3); // 4
objD.methode2(3); // 4
objD.methode2(2, 5); // 7
// Il n'est pas possible d'écrire : objB.methode2(2, 5); car cette
     méthode n'est définie que dans la classe D
```

4.2 Classes abstraites

Une classe abstraite peut posséder une ou plusieurs méthodes abstraites. Elle doit être déclarée abstraite. Une classe abstraite ne peut pas être instanciée.

Par exemple, nous pouvons déclarer une classe **Fruit** comme une classe abstraite.

```
public abstract class Fruit {
   // Attributs, constructeurs, méthodes

   // Déclaration d'une méthode abstraite
   public abstract String getForme();
}
```

Cette classe ne peut pas être instanciée, c'est-à-dire qu'il n'est pas possible d'écrire **Fruit f = new Fruit();**. Par contre, on peut écrire des classes qui héritent de cette classe abstraite, et ces classes filles pourront être instanciées (si elles ne sont pas déclarées abstraites). Les classes filles doivent impérativement définir toutes les méthodes déclarées abstraites dans la classe mère. Par exemple, on peut définir les classes **Banane** et **Orange**, qui sont des fruits, de la manière suivante :

```
public class Banane extends Fruit {
   // Attributs, constructeurs, méthodes

   // Implémentation de la méthode abstraite
   @Override
   public String getForme() {
      return "Longue et incurvée";
   }
}
```

```
public class Orange extends Fruit {
   // Attributs, constructeurs, méthodes

   // Implémentation de la méthode abstraite
   @Override
   public String getForme() {
      return "Ronde";
   }
}
```

Une classe abstraite sert à définir des concepts incomplets et factorise des attributs et/ou des comportements communs à un ensemble de sous-classes.

Une *interface* est une classe dont toutes les méthodes sont abstraites, et tous les attributs sont constants (déclarés **final**). Une interface est une spécification formelle de classe : on précise ce que les sous classes doivent offrir comme service en leur laissant la liberté de l'implémentation.