Travaux Dirigés n°4

Java Avancé

—M1 Apprentissage—

Héritage and co.

Héritage, redéfinitions etc.

▸ **Exercice 1.**

1. Écrire une classe `Car` représentant une voiture, avec deux champs `final` privés `brand` (chaine de caractère) et `value` (long).
2. Écrire le constructeur prenant en paramètre les valeurs des champs.
3. Faire un sorte qu'il soit impossible de créer une voiture avec une valeur négative.
4. Écrire les méthodes get pour les deux champs (eclipse peut le faire automatiquement via clic droit/source/generate getters).
5. Écrire une méthode `toString()` pour afficher une voiture et ses caractéristiques.
6. Écrire une classe `Garage` pour stocker des instances de `Car` (réfléchir à la structure de données adapté pour cela). Ajouter une méthode dans `Garage` pour ajouter une voiture dans le garage. Faire en sorte qu'il soit impossible d'ajouter une voiture `null` dans le garage à l'aide de `Objects.requireNonNull()`.
7. On veut que chaque garage ait un identifiant unique propre à chaque garage (faire aussi une méthode `getId()`). Pour l'exercice, on utilise le nombre d'instances crées de la classe `Garage`. (penser aux champs statiques).
8. Écrire une méthode `toString()` pour afficher un garage proprement à l'aide d'un `StringBuilder`.
9. Ajouter une méthode permettant de calculer la valeur d'un garage (somme de la valeur des voitures qu'il contient).
10. Écrire une méthode `firstCarByBrand` qui prend en paramètre une marque et retourne la première voiture de cette marque. Que doit-on faire s'il n'y a pas de voiture de cette marque ?
11. Les voitures peuvent maintenant avoir un niveau de vétusté. 0 est non vétuste, puis pour chaque niveau, la valeur de la voiture décroit de 1000e. Par exemple, une voiture valant 10 000e avec un niveau de vétusté 2 vaut mantenant 8 000. Ajouter un champ correspondant à ce niveau de vétusté et modifier l'implémentation en conséquence. Le niveau de vétusté peut être spécifié ou non à la construction (surcharge du constructeur).

12. Testez votre implémentation à l'aide des tests JUnit suivants : CarTest GarageTest.

▸ Exercice 2.

1. Exécuter le code suivant :

```
Car a = new Car("Audi",10000);
 Car b = new Car("BMW",9000);
 Car c = new Car("BMW",9000);
 Car d = a;

 System.out.println(a==b);
 System.out.println(b==c);
 System.out.println(a==d);
 System.out.println(a.equals(b));
 System.out.println(b.equals(c));
 System.out.println(a.equals(d));
```

   Le comportement est-il naturel ? Ajouter une méthode ayant pour signature `boolean equals(Car c);`.

2. Exécuter le code suivant :

```
ArrayList<Car> list = new ArrayList<>();
list.add(a);
list.add(b);

System.out.println(list.indexOf(a));
System.out.println(list.indexOf(b));
System.out.println(list.indexOf(c));
System.out.println(b.equals(c));
```

   Est-ce un comportement logique ? Lire la doc de `indexOf` de `List` et modifier votre code en conséquence.

3. Exécuter le code suivant :

```
HashSet<Car> set = new HashSet<Car>();
set.add(b);
System.out.println(set.contains(c));
```

   Est-ce un comportement logique ? Lire la doc de `contains` de `Set` et modifier votre code en conséquence.

4. Écrire une méthode `remove` dans Garage qui prend une voiture en argument et qui permet de retirer une voiture du garage.
5. Testez votre implémentation à l'aide des tests JUnit suivants : GarageTest2 CarTest2

▸ Exercice 3.

On veut pouvoir ajouter d'autres types de véhicules dans le garage.

1. Écrire une classe `Bike` contenant uniquement un champ concernant la marque.

2. On considère que la valeur d'un vélo est unique et fixé (constante) à 100 euros. Apporter les modifications nécessaires.
3. Quels changements doivent être effectués pour pouvoir ajouter des vélos dans le garage ? Le faire !
4. Le garage souhaite maintenant pouvoir faire des soldes pendant un certain temps. Créer une classe `Discount` comportant un champ value. Cette valeur doit remplacer le prix original du véhicule si le véhicule est en solde (champ Discount à null ou non).
5. Dans garage, créer une méthode `void protectionism(String brand);` qui retire tous les véhicules de la marque passé en argument, en parcourant une seule fois la liste. Tester, il est probable que votre première idée ne fonctionne pas...
6. Testez votre implémentation à l'aide des tests JUnit suivants : GarageTest3 DiscountTest BikeTest

▸ **Exercice 4.**

On veut pouvoir tester si le contenu de deux garages est le même.

Lancer le test GarageTest4

1. Pourquoi le test est-il un échec ?
2. Écrire une méthode `equals` pour `Garage` qui fait simplement appel à la méthode `equals` d'`ArrayList` (celle du contenu du garage). Le résultat est-il satisfaisant ?
3. Modifier le code afin de trier la liste (en utilisant la méthode `sort` de la classe `Collections`) au niveau de l'ajout d'un véhicule dans le garage (après chaque ajout) afin que le contenu du garage soit toujours trié. Pour pouvoir utiliser `sort`, il faut pouvoir comparer un véhicule à un autre. Ces critères de comparaison sont-ils satisfaisant ?
   — L'ordre alphabétique selon le nom du véhicule.
   — L'ordre alphabétique sur le nom de la marque.
   — Une combinaison entre le nom du véhicule et de sa marque ?
4. Quel est le nombre d'opérations dans le pire cas (s'il y a n véhicules dans les garages) dans le cas de n ajouts suivit d'un appel à equals dans les 2 solutions suivantes :
   (a) Tri de la liste au moment de la comparaison.
   (b) Insertion du véhicule à la bonne place.

   L'implémenter.
5. Que se passe-t-il pour le nombre d'opérations si les `equals` sont appelés entre chaque ajout ?
6. Que faire si on utilise des `LinkedList` plutôt que des `ArrayList` ?