Correction TD n°4

Java Avancé

—M1 Apprentissage—

Héritage and co.

Héritage, redéfinitions etc.

▸ **Exercice 1.**

1. Écrire une classe `Car` représentant une voiture, avec trois champs `final` privés `brand` (chaine de caractères), licencePlate (chaine de caractères) 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 trois 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 `add` 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 `getValue` 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 10 000 euros avec un niveau de vétusté 2 vaut maintenant 8 000 euros. 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). Veillez à ne pas dupliquer le code ! (faire appel à un constructeur dans un constructeur !)
12. Testez à l'aide des tests JUnit suivants : CarTest GarageTest.

```java
public class Car {
  private final String brand;
  private final int value;
  private final int vetuste;
  private final String licencePlate;

  public Car(String brand, String licencePlate, int value) {
    this(brand, licencePlate, value, 0);
  }

  public Car(String brand, String licencePlate, int value, int collection) {
    if(value < 0) throw new IllegalArgumentException("negative value");
    this.licencePlate = Objects.requireNonNull(licencePlate);
    int computedValue = value - collection*1000;
    if(computedValue*2 < value) throw new IllegalArgumentException("too old");
    this.brand = brand;
    this.value = computedValue;
    this.vetuste = collection;
  }

  public String getLicencePlate() {
    return licencePlate;
  }

  public String getBrand() {
    return brand;
  }

  public int getValue() {
    return value;
  }

  @Override
  public String toString() {
    return "Voiture " + getBrand() + " " + licencePlate + " " + value;
  }
}
```

```java
public class Garage {
  private final ArrayList<Car> list = new ArrayList<Car>();
  private final int id;
  private static int NB = 1;

  public Garage() {
    this.id = NB++;
  }

  public void addCar(Car car) {
    list.add(car);
  }

  @Override
  public String toString() {
    StringBuilder sb = new StringBuilder();

    sb.append("Garage id " + id);

    for(Car c: list) {
      sb.append(c.toString()).append("\n");
    }

    return sb.toString();
  }

  public int getValue() {
```

```
    int val = 0;
    for(Car c: list) {
      val += c.getValue();
    }
    return val;
  }
}
```

▸ Exercice 2.

1. Exécuter le code suivant :

```
Car a = new Car("Audi", "41abd75", 10000);
 Car b = new Car("BMW","42abc75", 9000);
 Car c = new Car("BMW","42abc75", 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` (ou faire control+clic gauche) 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 ? Un `HashSet` utilise un hashCode...

4. Écrire une méthode `remove` dans Garage qui prend une voiture en argument et qui permet de retirer **une** voiture du garage. Si la voiture n'existe pas, elle devra lever une IllegalStateException. Attention, on ne veut pas parcourir plusieurs fois la liste.

5. Testez à l'aide des tests JUnit suivants : GarageTest2 CarTest2

1. Lors de l'exécution de ce code le résultat est le suivant :
   false false true false false true
   Les objets b et c ne référent pas au même objet. Ce qui explique le résultat obtenu.
   Le résultat souhaité serait d'avoir false false true false true true ... Pour ce faire, on voudrait que la méthode equals() dise que 2 oranges sont égales si elles ont les mêmes champs :
2. Le résultat est 0 1 -1 true. Comme la dernière réponse est true (b et c sont égaux), on pourrait s'attendre à ce que c soit également trouvé dans la liste... En fait, la méthode indexOf(Object o) (qui renvoit la position de la première occurrence de l'objet au sein de la liste courante, ou -1 s'il n'a pu être trouvé) n'appelle pas notre méthode equals(Orange) :
   int indexOf(Object o)
   En effet, la méthode indexOf utilise la méthode equals(Object o).
   Par conséquent, il faut redéfinir cette dernière comme suit (en précisant la redéfinition) :
3. Set a besoin de HashCode

```
1  public boolean equals(Car c) {
2 //primitif first
3    return value == c.value && getBrand().equals(c.getBrand()) && c.getLicencePlate().equals(licencePlate);
4  }
5
6  @Override
7  public boolean equals(Object obj) {
8    if(!(obj instanceof Car)) return false;
9    return equals((Car)obj);
10 }
11
12 @Override
13 public int hashCode() {
14 return 42;
15 }
```

▸ **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 et sans créer de nouvelle liste. Tester, il est probable que votre première idée ne fonctionne pas...

6. Testez à l'aide des tests JUnit suivants : GarageTest3 DiscountTest BikeTest

✂ -

1. .
2. Champ static
3. Interface Vehicule.
4. Passer en classe abstraite, remonter la brand qui est commun. Ajouter un champ discount à null éventuellement. Faire une méthode getPrice dans Vehicule qui va gérer le discount et faire appel au getPrice de chacun grace à une méthode abstraite forcée. Attention aux constructeurs aussi.
5. Iterateur.

```java
public class Discount {
  private final int val;
  public Discount(int val) {
    this.val = val;
  }

  public int getVal() {
    return val;
  }
}
```

```java
public class Garage {
  private final ArrayList<Vehicule> list = new ArrayList<>();
  private final int id;
  private static int NB = 1;

  public Garage() {
    this.id = NB++;
  }

  public void addVehicule(Vehicule car) {
    list.add(car);
  }

  @Override
  public String toString() {
    StringBuilder sb = new StringBuilder();

    sb.append("Garage id " + id);

    for(Vehicule c: list) {
      sb.append(c.toString()).append("\n");
    }

    return sb.toString();
  }

  public int getValue() {
    int val = 0;
    for(Vehicule c: list) {
      //val += c.getValue();
      val += c.getRealValue();
    }
    return val;
  }

```

```
  public void protectionism(String brand) {
    Iterator<Vehicule> it = list.iterator();
    while(it.hasNext()) {
      Vehicule v = it.next();
      if(v.getBrand().equals(brand)) {
        it.remove();
      }
    }
  }
}
```

```
public abstract class Vehicule {
  private Discount d;
  private final String brand;

  public abstract int getValue();

  public Vehicule(String brand, Discount d) {
    this.brand = brand;
    this.d = d;
  }
  public Vehicule(String brand) {
    this(brand, null);
  }

  public int getRealValue() {
    if(d==null) {
      return getValue();
    }
    else {
      return d.getVal();
    }
  }

  public String getBrand() {
    return brand;
  }

  public void setDiscount(Discount d) {
    this.d = d;
  }
}
```

```
public class Car extends Vehicule {

  private final int value;
  private final int vetuste;
  private final String licencePlate;

  public Car(String brand, String licencePlate, int value) {
    this(brand, licencePlate, value, 0);
  }

  public Car(String brand, String licencePlate, int value, int collection) {
    this(brand, licencePlate, value, collection, null);
  }

  public Car(String brand, String licencePlate, int value, Discount d) {
    this(brand, licencePlate, value, 0, d);
  }

  public Car(String brand, String licencePlate, int value, int collection, Discount d) {
    super(brand, d);
    if (brand == null) throw new NullPointerException();
    this.licencePlate = Objects.requireNonNull(licencePlate);
    if(value < 0) throw new IllegalArgumentException("negative value");
```

```java
    int computedValue = value - collection*1000;
    if(computedValue*2 < value) throw new IllegalArgumentException("too old");
    this.value = computedValue;
    this.vetuste = collection;
  }


  public int getValue() {
    return value;
  }

  @Override
  public String toString() {
    return "Voiture " + getBrand() + " " + value;
  }

  public boolean equals(Car c) {
    //primitif first
    return value == c.value && getBrand().equals(c.getBrand()) && c.getLicencePlate().equals(licencePlate);
  }

  @Override
  public boolean equals(Object obj) {
    if(!(obj instanceof Car)) return false;
    return equals((Car)obj);
  }

  @Override
  public int hashCode() {
  return 42;
  }
}
```

```java
public class Bike extends Vehicule {
  public Bike(String brand) {
    this(brand, null);
  }

  public Bike(String brand, Discount d) {
    super(brand, d);
  }

  @Override
  public int getValue() {
    return 100;
  }
}
```

▸ 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` ?

- -

1. Faux car test de référence si non redéfini.
2. Non, le equals test case i avec case i, si ordre d'insertion différent foutu.
3. Il faut redéfinir compareTo. Celui d'un Car va appeler celui de véhicule si c'est pas un Car puis comparer selon TOUS les champs (brand, value, discount, vetuste, etc) dans l'ordre pour que l'ordre d'insertion ne soit pas trompeur. Pareil pour Bike. Et ça remonte à Vehicule pour comparer entre véhicules différents.
4. Cas 1 : n add puis 1 equals : $O(n + nlogn)$. Case 2 : n add puis 1 equals : $O(n^2 + 1)$

```java
@Override
public boolean equals(Object o){
  if (!(o instanceof Basket))
    return false;
  Basket b = (Basket) o;
  Collections.sort(items); // O(nlog(n))
  Collections.sort(b.items); // O(nlog(n))
  return items.equals(b.items);
}

public void add(Fruit f) {
  ListIterator<Fruit> iterator = items.listIterator();
  while(iterator.hasNext()){
    if(iterator.next().compareTo(f) > 0){
      break;
    }
  }
  iterator.add(f); // ArrayList : O(2n)+reallocation; LinkedList : O(n)
}
```

```java
//dans Vehicule
@Override
public int compareTo(Vehicule v) {
  //pour eviter a avoir a creer une methode renvoyer le nom du vehicule
```

```java
  return this.getClass().getName().compareTo(v.getClass().getName());
}

Dans Car:
@Override
public int compareTo(Vehicule o) {
  if(!(o instanceof Car)) {
    //n'est pas un Car, a comparer avec un autre vehicule plus haut
    return super.compareTo(o);
  }
  Car c = (Car)o;
  return (getBrand()+licencePlate+value+getValue()+vetuste).compareTo(c.getBrand()+c.licencePlate+c.value+c.
       getValue()+c.vetuste);
}
```