# **Exo 1**

Voir les classes Java déposées sur le git dans les packages :

* fr.dauphine.javaavance.td3.exos1et2.src
* fr.dauphine.javaavance.td3.exos1et2.test

# **Exo 2**

Pour consulter le code, voir les classes Java déposées sur le git dans les packages :

* fr.dauphine.javaavance.td3.exos1et2.src
* fr.dauphine.javaavance.td3.exos1et2.test

1. Le coportement est naturel car le test == ou .equals ne renvoie true que si les objets sont exactement les mêmes c’est-à-dire qu’ils doivent avoir le même code de hashage. Si deux objets ont exactement les mêmes valeurs pour chaque attribut mais n’ont pas le même code de hashage (car ce sont deux objets distincts), alors la méthode renvoie false.
2. Ce comportement est illogique car la voiture b ayant exactement les mêmes attributs que c a été ajoutée dans la liste. On aimerait donc avoir la position de b lorsqu’on demande list.indexOf(c). Pourquoi cette ligne renvoie -1 ? Car la méthode indexOf(Object o) parcourt la liste sur laquelle cette méthode est appelée et compare via la méthode equals si l’élément parcouru est égal à o. Si oui, la méthode indexOf renvoie l’indice i de l’élément parcouru égal à o et renvoie -1 sinon. Dans notre cas, on n’a pas redéfini la méthode equals de Object de telle sorte à ce que deux objets ayant exactement les mêmes attributs soient considétés comme égaux mais simplement créé une méthode de signature boolean equals (Car c). Il faut donc redéfinir equals de Object.
3. De la même manière que indexOf(Object o) boucle sur une List et teste si l’élément parcouru est égal à o via equals, la méthode contains(Object o) boucle sur un Set et teste si l’élément parcouru est égal à à via equals pour déterminer si oui ou non le Set contient o. Maitenant que la méthode equals de Object a été redéfinie dans Car (dans la question précédente), il suffit simplement de redéfinir la méthode hashCode pour que le comportement du code soit logique et qu’il renvoie true à set.contains(c) bien que seul b ait été ajouté dans le set. En effet, b et c ont exactement les mêmes attributs et c’est la raison pour laquelle on leur attribue le même code de hashage.

# **Exo 3**

Dans cet exercice, on va redéfinir les deux classes Garage et Car en y ajoutant d’autres classes, avec notamment Bike et Vehicle. Le principe de résolution de l’ensemble des questions repose sur l’héritage et le polymorhisme. En effet, désormais, les classes Bike et Car sont des classes-filles de Vehicle.

Voir les classes Java déposées sur le git dans les packages :

* fr.dauphine.javaavance.td3.exo3.src
* fr.dauphine.javaavance.td3.exo3.test

# **Exo 4**

Les classes Java déposées sur le git pour cet exercice se trouvent dans les packages :

* fr.dauphine.javaavance.td3.exo4.src
* fr.dauphine.javaavance.td3.exo4.src

1. Le test est un échec notamment parce que les arrayList stockant les véhicules du garage1 et du garage2 ne sont pas égaux. En effet, ils stockent exactement les mêmes véhicules mais pas dans le même ordre.
2. Non avec la méthode ci-dessous, le résultat n’est toujours pas satisfaisant.

@Override

**public** **boolean** equals(Object obj) {

Garage other = (Garage) obj;

**return** Objects.*equals*(**this**.myGarage, other.myGarage);

}

1. Si on modifie le code afin de trier la liste au niveau de l’ajout d’un véhicule dans le garage, on sait que pour pouvoir utiliser sort, méthode de Collections permettant de trier une liste, on doit choisr un critère de comparaison pour comparer le véhicule à un autre. Pour que le contenu des deux listes garage1 et garage2 soient le même, c’est-à-dire que leurs éléments soient triées et apparaissent donc dans le même ordre, il faut combiner plusieurs critères de comparaison qu’on teste successivement dans l’ordre suivant. En effet, comparer sur la valeur du véhicule et/ou sa marque ne suffisent pas car les véhicules sont également définis par d’autres attributs. On va donc comparer sur les critères suivants :

* Nom de la classe (Bike ou Car)
* Marque – brand
* Value
* Discount
* Obsolescence (pour les objets Car uniquement 🡪 redéfinition de compareTo dans Car)

1. a- Si les éléments ne sont pas triés, il faut donc un appel à sort après les ajouts d’élément. Sachant qu’on ajout n éléments dans le garage, le nombre d’opérations est de l’ordre O(nlog(n))

b- Si les éléments sont déjà triés et qu’on ajout n éléments dans le garage, le nombre d’opérations est de O(n)

1. Si les equals sont appelés entre chaque ajout d’élément, le nombre d’opérations vaut :
2. Contrairement à l’ArrayList qui utilise un tableau dynamique pour stocker les éléments, une LinkedList utilise une liste doublement liée pour stocker les éléments. Les LinkedList consomment donc plus d’espace que les ArrayList : il est nécessaire de stocker le nœud précédent et le nœud suivant pour chaque élément (signification de ‘doublement liée’). Son avantage repose par ailleurs dans cette particularité car la manipulation des éléments se fera plus rapidement, la liste étant doublement liée, aucun décalage de bit n’est requis en mémoire.