Equals - Collections

Mathieu Bourgais
mathieu.bourgais@insa-rouen.fr

2023

Préambule

Vous pouvez utiliser l'éditeur de texte de votre choix. Vous pouvez aussi utiliser un IDE comme Eclipse, Visual Studio Code ou encore Intellij, qui présente de nombreux avantages (auto-complétion, par exemple). Dans Eclipse, vous devez toujours travailler dans un projet (à créer au début de chaque TP/mini-projet) dans lequel vous définirez vos classes.

Pour rappel, pour compiler et exécuter un code source Java, vous pouvez utiliser votre éditeur favori, ou ouvrir un terminal puis naviguer jusqu'au répertoire désiré, et taper :

— javac monFichier.java
— java monFichier

1 Exercice 1 - Equals et Hashcode

En Java, le comportement de l'opérateur == dépend du type des variables. En effet, pour les types simples (i.e. les boolean, les int, les float, etc), l'opérateur == compare l'égalité des valeurs des variables. En revanche, pour les types objets (i.e. les tableaux, les String, les instances de classes, et tout le reste), l'opérateur == compare l'égalité des références. Ainsi, deux instances différentes d'une même classe ne pourront pas être ==, même si les valeurs de tous leurs attributs sont égales. Pour tester l'égalité des valeurs, sur des variables de type objet, Java propose alors d'utiliser la méthode **equals(Object o)**.

1.1 Premier test

Récupérez les interfaces et classes **Compte**, **CompteBancaire**, **CompteCourant** et **CompteEpargne** développées lors du TD2, ainsi que la classe **Personne** fournie dans les sources de ce TD3. Compilez l'ensemble de ces classes, ainsi que la classe **TestEquals** fournie, puis exécutez le programme. Observez les résultats. Ceux-ci vous semblent-ils cohérents ? Pourquoi ?

1.2 Premières modifications

Par défaut, l'implémentation de la méthode **equals** retourne true uniquement si les deux objets comparés ont même la même référence, et donc, s'ils sont ==. En d'autres termes, c1.equals(c2) $\iff$ c1 == c2 par défaut. Ainsi, il est donc bien souvent nécessaire de redéfinir la méthode **equals** au sein des classes développées, afin de permettre la comparaison de deux objets en fonction des valeurs de leurs attributs. Pour cela, au sein de la méthode **equals**, il est alors nécessaire de vérifier que les deux objets comparés sont bien de la même classe, puis de tester ensuite l'égalité de l'ensemble de leurs attributs.

Lorsqu'une classe redéfinit la méthode **equals**, par convention, elle doit également redéfinir la méthode **hashCode()**. Cette méthode permet d'obtenir un résumé d'une instance d'une classe, sous forme d'une valeur de type int. Cette méthode doit être redéfinie afin d'optimiser les algorithmes de recherche au sein d'ensembles (que vous manipulerez pendant l'exercice 2). Ces algorithmes utilisent en effet les méthodes hashCode et equals afin de trouver l'instance d'un objet recherché au sein d'un ensemble. Ainsi, si deux variables sont equals, leurs hashCode doivent également être identiques.

Modifiez votre classe **CompteBancaire**, afin d'en redéfinir la méthode **equals** (ainsi que la méthode **hashCode**). Vous pouvez, pour cela, vous inspirer des méthodes **equals** et **hashCode** de la classe **Personne** fournie.

Compilez de nouveau vos classes, ainsi que la classe **TestEquals** fournie, et vérifiez la validité de l'ensemble des tests. Si besoin est, effectuez les modifications nécessaires (dans votre code ;)) jusqu'à ce que l'ensemble des tests, hormis le premier, retourne true.

Compilez la classe **TestEqualsComptes** fournie, puis exécutez le programme. Observez les résultats. Ceux-ci vous semblent-ils cohérents ? Pourquoi ?

1.3 Secondes modifications

Modifiez vos classes **CompteCourant** et **CompteEpargne** afin d'en redéfinir les méthodes **equals** (ainsi que les méthodes **hashCode**). Pour information, au sein d'une classe fille, les méthodes equals et hashCode de la classe mère peuvent être invoquées à l'aide d'un appel à super.equals ou à super.hashCode. Compilez de nouveau vos classes, ainsi que la classe **TestEqualsComptes** fournie, et vérifiez la validité de l'ensemble des tests.

Comme précédemment, effectuez les modifications nécessaires jusqu'à ce que l'ensemble des tests soit cohérent (ici, tous sauf les deux derniers doivent retourner false).

2 Exercice 2 - Collections

Les collections permettent de gérer des groupes d'objets de même type, dont le nombre n'est pas forcément connu à l'avance. Contrairement aux tableaux,

les collections sont des structures dynamiques, et sont donc redimensionnables. Cependant, elles sont également plus coûteuses en termes de mémoire.

Avec les collections, il est possible de choisir librement l'interface et l'implémentation désirées. Plus précisément, une multitude de collections ayant une interface commune **List** peuvent être implémentées au sein de plusieurs classes différentes, ayant chacune leurs spécificités. On peut ainsi, par exemple, avoir une classe **ArrayList**, implémentant les listes sous forme de tableaux, et une classe **LinkedList**, implémentant les listes sous forme de listes chaînées. Les choix d'implémentation peuvent donc avoir un impact sur la complexité (notamment en temps) des opérations. Cependant, quelle que soit l'implémentation choisie, les fonctionnalités et méthodes définies dans l'interface **List** doivent être proposées.

En Java, les collections sont génériques. C'est à dire qu'elles prennent en paramètre le type des objets qu'elles contiennent. Cela permet de définir une et une seule fois le code des collections, sans avoir besoin de le réécrire à nouveau à chaque fois que le type contenu change (i.e. éviter d'avoir un ensemble de collections pour les int, un ensemble de collections pour les float, un ensemble de collection pour les String, ...).

Pour manipuler une collection, il est alors nécessaire de définir son interface, son implémentation et le type des objets manipulés. Par exemple, la déclaration d'une variable représentant un ensemble de String, implémenté à l'aide d'une table de hachage, se fait comme suit : **Set<String> s = new HashSet<String>() ;**. Pour une variable représentant une liste de Personne, implémentée à l'aide d'un tableau, on écrirait : **List<Personne> l = new ArrayList<Personne>() ;**.

2.1 Création de la banque

Le code de la classe **Banque**, implémentant l'interface **BanqueInt**, est incomplet. En particulier, toutes les opérations liées à la manipulation des ensembles sont manquantes. Le comportement attendu des différentes méthodes manquantes est cependant spécifié au sein de l'interface **BanqueInt**. Vous pouvez vous référer à la documentation sur les ensembles, disponible à l'adresse suivante : *https ://docs.oracle.com/javase/8/docs/ api/java/util/Set.html*, afin de vous aider. Implémentez les méthodes manquantes.

Compilez votre classe Banque ainsi que la classe **TestBanque** fournie. Vérifiez le bon fonctionnement de votre code. Pourquoi la banque ne contient-elle que trois comptes à la fin de l'exécution ?

2.2 Ajustements liés à la banque

Modifiez la classe **Personne** fournie afin de lui ajouter un nouvel attribut représentant une **List** de **Compte**. Vous penserez à ajouter les accesseurs et mutateurs de ce nouvel attribut, et à modifier / ajouter les constructeurs nécessaires

afin de l'initialiser convenablement. Vous ajouterez également une méthode **void ajouterCompte(Compte compte)** permettant d'ajouter un nouveau compte à la liste, s'il n'est pas déjà présent, ainsi qu'une méthode **void supprimerCompte(Compte compte)** permettant de supprimer un compte de la liste, s'il y est présent. Vous utiliserez l'implémentation ArrayList pour les List. Vous pouvez vous référer à la documentation sur les listes, disponible à l'adresse suivante : *https ://docs.oracle.com/javase/8/docs/api/java/util/List.html*, afin de vous aider.

Modifiez la classe **Banque**, et plus particulièrement les méthodes **ajouterCompte(CompteCourant compte)**, **ajouterCompte(CompteEpargne compte)**, **supprimerCompte(CompteCourant compte)** et **supprimerCompte(CompteEpargne compte)**, afin que, lors de l'ajout ou de la suppression d'un compte au sein d'une Banque, celui-ci soit également ajouté à / supprimé de la List des comptes de la Personne à laquelle il est associé.

Compilez vos classes, ainsi que la classe **TestBanqueComplet** fournie, et vérifiez le bon fonctionnement de votre code.