Les collections (2)

DUT M3105 : Conception et programmation objet avancées

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2016-2017

De l'identité des objets

- Dans les listes ou les `HashSet`, des méthodes comme `contains` ou `remove` considèrent que deux objets a et b sont égaux ssi `a.equals(b)` (et `b.equals(a)`) ;
- `equals` est définie dans la classe `Object` ;
- le `equals` implémenté dans `Object` dit que deux objets sont égaux s'ils ont la même *adresse* ;
- on peut *redéfinir* `equals`. Pour `String`, deux strings sont égales si elles contiennent les mêmes caractères.
- comment choisir si oui ou non on redéfinit `equals` ?

Les classes « entités »

Definition

Un objet est une entité s'il représente une *individualité* unique, dont les propriétés peuvent éventuellement varier, sans remettre en cause l'identité de l'objet.

- Par exemple, dans un jeu, si l'objet représentant le magicien `Rincewind` perd deux points de vie, il continue quand même à représenter le même personnage ;
- si on a deux objets représentant Rincewind, c'est gênant, il faut les synchroniser ;
- deux objets « sac de pièce d'or » peuvent contenir les même quantités d'or (donc avoir la même représentation interne) et reste distincts.

L'identité d'un objet entité est son **adresse.**
La méthode `equals` de `Object` est donc adaptée à ce cas.

Les classes « valeurs »

Definition

Un objet qui représente une *valeur* est entièrement défini par ses propriétés, qui sont immuables.

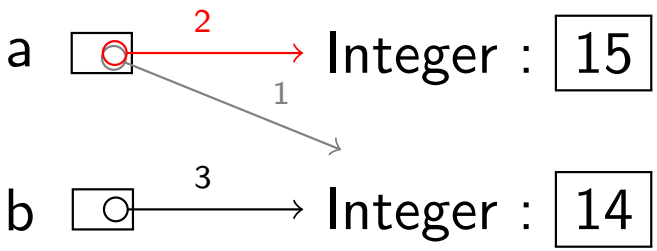
- Par exemple, un objet `Heure` représentant l'heure 3h10 est entièrement défini par ces deux nombres, 3 et 10 ;
- deux objets heures valant 3h10 seront donc égaux ;
- une fois l'objet créé, il ne doit pas être modifié (il n'a pas de setters, par exemple) ;
- en java : c'est le cas des classes-sœurs des types primitifs (Integer, Character...) ; c'est aussi le cas des `String`. ;
- un objet-valeur peut être partagé sans risque ;

Deux objets valeurs avec le même contenu sont égaux.
Il faut alors redéfinir les méthodes `equals` et `hashCode`.
(sauf pour les enums, parce que `enum` garantit l'unicité de l'instance).

Objets et références

Ne pas confondre objets et références. Si un objet-valeur ne peut changer, une référence vers un objet-valeur peut parfaitement être modifiée !

```
1 Integer a= new Integer(14); // 1
2 Integer b= a; // 2
3 a= new Integer(a.intValue() + 1); // 3
```



- Aucun des deux objets de classe Integer n'est variable ;
- le contenu de la variable a, qui contient une adresse (une référence) vers un `Integer` change

Et entre les deux ?

- une partie des variables d'instances définissent l'identité de l'objet ;
- par exemple le code d'un produit (constant), alors que son prix peut changer ;
- c'est utilisé assez souvent...
- mais est-ce une bonne idée ? doit-on avoir en mémoire deux objets pour le même produit ?
- se rencontre néanmoins assez souvent dans les programmes.

Excursus : les tables de hachage

(voir les autres transparents, vu que dessiner des figures c'est quand même une plaie).

Equals et HashCode

- pour que tout fonctionne, **il faut que equals et hashCode soient compatibles**
- par défaut, les classes héritent du code de Object ;
- celui-ci définit equals et hashCode en fonction de *l'adresse* des objets ;
- Pour redéfinir la relation d'égalité d'une classe, on doit obligatoirement redéfinir à la fois **equals** et **hashCode**
- en respectant la règle :
- *si deux objets sont égaux pour `equals`, alors ils ont le même hashCode.*
- si `a.equals(b)`, alors `a.hashCode() == b.hashCode()` ;
- la réciproque n'est pas nécessaire (deux objets différents *peuvent avoir le même hashCode* ;

Equals et HashCode, marche à suivre

- choisir les propriétés de l'objet qui seront utilisées ;
- pour equals, comparer les propriétés entre les deux objets concernés ;
- pour hashCode, utiliser tout ou partie des propriétés choisies.

Écriture de equals

- argument de `equals` : un `Object` ;
- il faut vérifier son type.

```
class Coordonnee{
  int x, y;
  ...
  @Override
  public boolean equals(Object o) {
    if (o instanceof Coordonnee) {
      Coordonnee autre= (Coordonnee) o;
      return this.x== autre.x
             && this.y == autre.y;
    } else {
      return false;
    }
  }
}
```

Conseils pour equals

- quand une propriété est elle-même un objet, on peut utiliser sa méthode `equals` pour la comparer :

```
1  class Coordonnee {
2    Integer x, y;
3    ...
4    @Override
5    public boolean equals(Object o) {
6      if (o instanceof Coordonnee) {
7        Coordonnee autre= (Coordonnee) o;
8        return this.x.equals(autre.x)
9               && this.y.equals(autre.y);
10     } else {
11       return false;
12     }
13   }
14 }
```

- au lieu de `instanceof`, on compare parfois `o.getClass()` et `this.getClass()`.

@Override

- Annotation java, pas obligatoire mais très utile ;
- le compilateur vérifie que la méthode est bien la redéfinition d'une méthode existante ;
- permet de détecter des bugs comme :

```
1     @Override
2     public boolean equals(Coordonnee autre) {
3         Coordonnee autre= (Coordonnee) o;
4         return this.x.equals(autre.x)
5                 && this.y.equals(autre.y);
6     }
```

- sans `@Override`, ce code compile...
- mais ne redéfinit pas correctement le `equals` de `Object` ;
- en fait, redéfinit une méthode qui a le même nom, mais pas la même signature ;
- et qui n'est pas appelé par les collections !

Redéfinition de hashCode

- hashCode doit renvoyer un entier calculé à partir des propriétés utilisées dans `equals` ;
- il vaut mieux que le hashCode *disperse* les valeurs, les ventile façon puzzle.
- *typiquement*, s'il y a trois propriétés p_1, p_2, p_3, on calculera le hashCode de chacune, h_1, h_2, h_3, et on les combinera de la manière suivante :

$$h_1 + 31 * h_2 + 31 * 31 * h_3$$

(on peut remplacer 31 par un autre nombre premier)

Redéfinition de hashCode

Déterminer le hashcode des propriétés :

- un objet quelconque a un hashCode qu'on peut utiliser ;
- les classes associées au types primitifs disposent de méthodes statiques `hashCode(v)`.

```
class Demo {
  private int a;
  private String s;
  private double d;
  ...
  @Override
  public int hashCode() {
    return a+31*(s.hashCode()
           + 31*Double.hashCode(d));
  }
}
```

attention à prendre en compte le cas de propriétés qui peuvent être à null.

Ensembles et Maps ordonnées

- les éléments sont listés dans un ordre significatif pour le lecteur humain (ordre alphabétique par exemple) ;
- sur `SortedSet`, on dispose de méthodes comme `tailSet(e)` qui permettent d'accéder à tous les éléments supérieurs ou égaux à `e`.

Comparaison de `HashSet<String>` et `TreeSet<String>` :

```
List<String> l= Arrays.asList("hvd",
                "dfz","adf", "acs", "bxs");
HashSet<String> s1= new HashSet<>(l);
TreeSet<String> s2= new TreeSet<>(l);
System.out.println("hashset : "+ s1);
System.out.println("treeset : "+ s2);
```

affiche :

```
hashset : [acs, adf, hvd, dfz, bxs]
treeset : [acs, adf, bxs, dfz, hvd]
```

Ensembles et maps ordonnés

Pour pouvoir construire un ensemble trié, il faut disposer d'un ordre de tri. Cet ordre peut être :

- défini au niveau de la classe des éléments en implémentant l'interface `Comparable` : c'est une décision importante, qui indique que les éléments de cette classe sont *a priori* ordonnés de cette façon là ;
- défini par un objet extérieur, un `Comparator`, séparé de la classe des éléments.
  - permet d'ordonner des éléments d'une classe qu'on ne peut modifier ;
  - permet d'utiliser un ordre différent de l'ordre par défaut.

L'interface Comparable

Pour une classe `A`, implémenter l'interface `Comparable<A>` signifie fournir une méthode

```
int compareTo(A autre)
```

qui comparera `this` à `autre`.

- si `this` égal à `autre`, `compareTo` doit renvoyer 0 ;
- si `this` $>$ `autre`, `compareTo` doit renvoyer un nombre positif ;
- si `this` $<$ `autre`, `compareTo` doit renvoyer un nombre négatif ;

Il est fortement conseillé que `compareTo` soit compatible avec equals : a.equals(b) $\Leftrightarrow$ a.compareTo(b) == 0

Comment ordonner des éléments

- Dans la plupart des cas, on va comparer séparément les variables d'instance, qui sont généralement elles-même Comparables.
- Par exemple : pour des heures, on comparera d'abord par heures, puis, à heures égales, on comparera les minutes.
- on choisit donc l'ordre dans lequel on compare les variables d'instance.
- pour une variable `a`, on compare `this.a` et `autre.a`. S'ils ne sont pas égaux, l'un est plus grand que l'autre et on a terminé ;
- sinon, on regarde la variable suivante ;
- pour la dernière variable, si on a l'égalité, alors on retourne 0.

Exemple

Version très mécanique de CompareTo (en utilisant la méthode `compare` disponible dans la classe `Integer`).

```
public class Temps implements Comparable<Temps> {
  int heures, minutes, secondes;

  public int compareTo(Temps autre) {
   int result= Integer.compare(this.heures, autre.heures);
   if (result != 0) return result;
   result= Integer.compare(this.minutes, autre.minutes);
   if (result != 0) return result;
   result= Integer.compare(this.secondes, autre.secondes);
   return result;
  }
}
```

Autre version

pour les **entiers,** la comparaison de a et b renvoie un résultat du même signe que (a - b).
On peut donc écrire :

```
public class Temps implements Comparable<Temps> {
  int heures, minutes, secondes;

  public int compareTo(Temps autre) {
   int result= this.heures - autre.heures;
   if (result != 0) return result;
   result= this.minutes - autre.minutes;
   if (result != 0) return result;
   result= this.secondes - autre.secondes;
   return result;
  }
}
```

Exemple avec des variables objets

On utilisera les méthodes `compareTo` des classes des variables d'instances.

```
/**
 * Indicatif téléphonique, ex. 33 pour France.
 */
public class Indicatif implements Comparable<Indicatif> {
   int code;
   String pays;
   ...
   public int compareTo(Indicatif autre) {
    int result= this.code - autre.code;
    if (result != 0) return result;
    result= this.nom.compareTo(autre.nom);
    return result;
   }
}
```

(exemple pas très réaliste : comme code et pays identifient le pays de manière unique, on n'a normalement besoin d'en regarder qu'un seul).

Et les autres cas ?

- l'ordre que nous venons d'utiliser est lexicographique : on regarde d'abord le premier champ, puis le second, etc... comme pour ordonner deux mots dans l'ordre alphabétique ;
- c'est très fréquemment le cas dans un contexte métier (pensez à l'ordre des lignes d'une feuille excel) ;
- on peut rencontrer des ordres plus complexes : par exemples, les rationnels p/q.

Utilisation pour des ensembles ordonnés

Si le type `A` est comparable, alors je peux créer :

- des `TreeSet<A>` ;
- des `TreeMap<A,K>` ;

Il suffit d'écrire :

```
TreeSet<A> s= new TreeSet<>();
```

Remarque

le `TreeSet` utilisera automatiquement la méthode `compareTo` pour comparer tout nouvel élément aux éléments déjà présents et l'insérer au bon endroit.

Comparator

- L'interface `Comparator` encapsule une méthode de comparaison, en la *séparant* de la classe comparée ;
- Elle permet d'utiliser plusieurs fonctions de comparaisons différentes.
- Le comparateur sera un objet à part, qui ne servira qu'à comparer.
- Exemple : comparaison d'indicatifs, par nom de pays d'abord et par numéro ensuite :

```
class CompIndicatifParNom
  implements Comparator<Indicatif> {
  @Override
  public int compare(Indicatif a, Indicatif b) {
    int r= a.pays.compareTo(b.pays);
    if (r!= 0) return r;
    return a.code - b.code;
  }
}
```

Utilisation d'un Comparator

On passe le comparator au constructeur de `TreeSet` ou de `TreeMap` :

```
// ensemble d'indicatifs triés par nom de pays :
TreeSet<Indicatif> s=
   new TreeSet<>(new CompIndicatifParNom());
s.add(new Indicatif(33, "France");
s.add(new Indicatif(1, "USA");
```

Construction d'un comparateur en java 8

- Java 8 facilite la construction de comparateur, grâce à la méthode statique `Comparator.comparing()`, qui prend comme argument un « extracteur de clef » (une méthode qui retourne la valeur d'un type de champ) ;
- cette méthode peut prendre comme argument une *référence de méthode*, qui donne le nom du getter à utiliser pour récupérer la propriété désirée.

L'exemple précédent en java 8 :

```
// ensemble d'indicatifs triés par nom de pays :
Comparator<Indicatif> c=
    Comparator.comparing(Indicatif::getPays)
              .thenComparing(Indicatif::getCode);
TreeSet<Indicatif> s= new TreeSet<>(c);
```

Ici, `Indicatif::getPays` désigne la méthode `getPays` de la classe `Indicatif`.

Comparator et ensembles

- supposons qu'un comparateur soit utilisé pour construire un ensemble ;
- si deux éléments a et b sont tels que `comparator.compare(a,b) == 0` , alors le comparateur considèrera qu'il s'agit du même élément ;
- dans un ensemble, ça signifiera que a et b ne seront pas distingués ;
- il faut donc que dans ce cas, le comparateur doit être compatible avec la méthode `equals` de la classe de a et b.
- **a.equals(b) $\Leftrightarrow$ comparator.compare(a,b) == 0**

Comparator et listes

- Les méthodes `Collections.sort` et `Arrays.sort` trient des listes et des tableaux ;
- Elles peuvent prendre comme argument un comparateur ;
- pour une liste, contrairement à un ensemble, l'ordre n'a pas besoin d'être *total* : a et b peuvent être différents, et `comparator.compare(a,b)` peut renvoyer 0.
- on n'a pas forcément besoin de la compatibilité d'un tel comparator avec equals.

Exemple : une liste d'étudiants, ordonnés par *note* :

```
Comparator<Etudiant> c=
    Comparator.comparing(Etudiant::getNote);
List<Etudiant> l= .....;
Collections.sort(l, c); // trie l par note.
```

documentation de sort :

> *This sort is guaranteed to be stable : equal elements will not be reordered as a result of the sort.*