TD n° 4
Licence Informatique (L2)
« Programmation objet avancée »

F. BERTRAND

Année universitaire 2020–2021

Éléments de correction

Concepts abordés :
- Notion d'interface
- Utilisation d'interfaces pour la comparaison d'objets

1 Implémentation d'une interface

La classe `java.util.Arrays` possède une méthode `sort()` :

```
1 public class Arrays
2 {
3   public static void sort(Object[] a) { ... }
4 }
```

La documentation indique que :

> « *Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the* `Comparable` *interface. Furthermore, all elements in the array must be mutually comparable (that is,* `e1.compareTo(e2)` *must not throw a* `ClassCastException` *for any elements* `e1` *and* `e2` *in the array).*
>
> *This sort is guaranteed to be stable : equal elements will not be reordered as a result of the sort.*
>
> *The sorting algorithm is a modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed* $n \times log(n)$ *performance.* »

Cette méthode permet de trier un tableau d'objets à **une condition** : la classe à laquelle appartiennent les objets doit implémenter l'interface `java.lang.Comparable` décrite ci-dessous :

```
1 public interface Comparable<T>
2 {
3   public int compareTo(T o);
4 }
```

Les commentaires associés à cette méthode sont les suivants :

« *Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. In the foregoing description, the notation* `sgn(`*expression*`)` *designates the mathematical* signum *function, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive. The implementor must ensure* `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` *for all x and y. (This implies that* `x.compareTo(y)` *must throw an exception if and only if* `y.compareTo(x)` *throws an exception)*

The implementor must also ensure that the relation is transitive :

`(x.compareTo(y)>0 && y.compareTo(z)>0)` *implies* `x.compareTo(z)>0`.

Finally, the implementer must ensure that `x.compareTo(y)==0` *implies that :* `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, *for all z.* »

1.1 Tri non paramétré

Écrire une classe `Point` comportant deux attributs : `x` et `y` et implémentant l'interface `Comparable`. La comparaison s'effectuera sur la distance du point par rapport à l'origine.

Dans un programme de test, créez un tableau de quelques objets `Point` différents et utilisez la méthode `sort()` pour trier ce tableau d'objets.

Éléments de correction

Pour rappel, implémenter une interface consiste à définir la (ou les) méthode(s) déclarées dans l'interface ici la méthode `compareTo` *déclarée dans l'interface* `Comparable`.

```
1  public class Point implements Comparable<Point>
2  {
3     private double x, y;
4
5     public Point (double abs, double ord)
6     {
7        this.x = abs;
8        this.y = ord;
9     }
10
11    public double donneX() { return this.x; }
12    public double donneY() { return this.y; }
13
14    public double distanceOrigine()
15    {
16       return Math.sqrt(this.x*this.x + this.y*this.y);
17    }
18
19    public int compareTo(Point p)
20    {
21       double distanceThis = this.distanceOrigine();
22       double distanceP = p.distanceOrigine();
23
24       if (distanceThis < distanceP)
25       {
26          return -1;
27       }
28       else
29       {
30          if (distanceThis > distanceP)
31          {
32             return 1;
33          }
34          else
35          {
```

```
            return 0;
         }
      }

      /* Le code ci-dessous (en commentaires) est représentatif
       * d'une fausse "bonne idéé car dangereux si la difference
       * depasse la capacite d'un entier

       return (int)(this.distanceOrigine() - p.distanceOrigine());

       */
   }

   public String toString()
   {
      return '(' + Double.toString(this.x) + ',' + Double.toString(this.y) + ')';
   }

}
```

```
import java.util.Arrays;

public class TestPoint
{
   public static void main (String[] args)
   {
      Point[] t = {
            new Point(1,2), new Point(2,3), new Point(2,1), new
            Point(4,5)
      };

      System.out.println(Arrays.asList(t));
      // tri du tableau
      Arrays.sort(t);
      System.out.println(Arrays.asList(t));

   }

}
```

1.2 Tri paramétré

On souhaite maintenant pouvoir trier les points soit selon leur abscisse ou soit selon leur ordonnée.

Une « mauvaise » approche serait de « bricoler » la méthode `compareTo` pour qu'elle puisse prendre en compte l'ordre souhaité (par exemple en ajoutant un attribut indiquant cet ordre dans la classe `Point`).

Heureusement, il est possible de trier des objets en fournissant le critère de tri au moment de l'appel de la méthode devant effectuer ce tri. Dans la bibliothèque Java, cela est réalisé grâce à deux méthodes `sort` :

1. Cette notation un peu particulière indique que le type utilisé par `Comparator` peut être `T` ou un super-type de `T`. Par exemple, si `T` était `Integer` alors `Comparator` fonctionnerait avec `Number` et `Object`.

```
public class Collections { // permet de trier des listes d'objets T
   ...
   public static <T> void sort(List<T> list, Comparator<? super T>[1]  c) { ... }
   ...
}

public class Arrays { // permet de trier des tableaux d'objets T
   ...
   public static <T> void sort(T array, Comparator<? super T> c) { ... }
   ...
}
```

Vous remarquerez que ces deux méthodes (de classe) utilisent un type générique `T` ce qui leur permet de trier n'importe quel type d'objet à condition que soit fourni un objet `c` implémentant l'interface `Comparator<T>` :

```
public interface Comparator<T> {
   int compare(T o1, T o2)
}
```

La valeur entière retournée suit les mêmes règles que pour celles données pour la méthode `compareTo` déclarée dans `Comparable`. Vous pourrez noter ici que, contrairement à `compareTo`, la méthode `compare` utilise une notation « fonctionnelle » car les deux objets à comparer apparaissent comme paramètres de la méthode.

Une fois les classes implémentant les critères de tri (et donc l'interface `Comparator`) créées , il est possible d'écrire le code suivant :

```
Point[] points = {new Point(1,2), new Point(2,3), new Point(2,1), new Point(4,5)};
TriPointAbscisse critere = new TriPointAbscisse();
Arrays.sort(points,critere); // tri des points par abscisse croissante
```

Écrire les classes `TriPointAbscisse` et `TriPointOrdonnée` qui implémenteront l'interface `Comparator`. Si les abscisses (ou les ordonnées) sont identiques alors la seconde coordonnée deviendra le second critère de tri. Par exemple, dans le code présenté ci-dessus le point (2,1) sera avant le point (2,3).

Éléments de correction

Seule la classe `TriPointAbscisse` est donnée car le code de `TriPointOrdonnée` est pratiquement identique.

```
import java.util.Comparator;
import java.util.Arrays;

public class TriPointAbscisse implements Comparator<Point> {
        public int compare(Point p1, Point p2) {
                if (p1.donneX() > p2.donneX()) {
                        return 1;
                }
                else if (p1.donneX() < p2.donneX()) {
                        return -1;
                }
                else if (p1.donneY() > p2.donneY()) { // x egaux
                           return 1;
                }
                else if (p1.donneY() < p2.donneY()) {
                        return -1;
```

```
                }
                else return 0; // x et y egaux
        }

   public static void main (String[] args)
   {
      Point[] t = {
            new
            Point(4,5),new Point(1,2), new Point(2,3), new Point(2,1),
      };

      System.out.println(Arrays.asList(t));
      // tri du tableau
      Arrays.sort(t,new TriPointAbscisse());
      System.out.println(Arrays.asList(t));

   }
}
```

1.3 Héritage et interfaces

On souhaite maintenant créer une classe `Marqueur` (au sens repère posé sur une carte) qui associe un point et un libellé (classe `String` qui elle-même implémente `Comparable`) utilisé lors de sa visualisation.

Donnez deux manières de définir cette classe `Marqueur`. On souhaite que les instances de `Marqueur` soit triées, par défaut, selon l'ordre alphabétique de leur libellé, comment procéder ?...

Éléments de correction

Deux solutions :
- *soit par héritage, et dans ce cas on peut dire que un marqueur **est un** point ;*
- *soit par composition, et dans ce cas on peut dire que un marqueur **a (possède) un** point ;*

```
// par héritage...
class Marqueur extends Point {
  private String libellé;
}

// par composition...
class Marqueur {
  private Point p;
  private String libellé;
}
```

Cependant avec l'approche par héritage, comme `Marqueur` hérite de `Point` qui elle-même implémente `Comparable<Point>`, on ne peut pas implémenter une seconde fois la même interface avec un type paramétrique différent :

```
// La déclaration de Marqueur donnée ci-dessous ne se compile pas...
// Le message d'erreur est le suivant :
// error : Comparable cannot be inherited with different
// arguments: <Marqueur> and <Point>
class Marqueur extends Point implements Comparable<Marqueur> {
  private String libellé;
  public int compareTo(Marqueur m) {
    return this.libelle.compareTo(m.libelle);
```

```
  }
}
```

Si on retire « `implements Comparable<Marqueur>` » alors la classe se compile mais ce sera la méthode `compareTo` de `Point` qui sera appelée.
La seule approche possible est celle par composition :

```
class Marqueur implements Comparable<Marqueur> {
  private Point p;
  private String libellé;
  public int compareTo(Marqueur m) {
    return this.libelle.compareTo(m.libelle);
  }
}
```

1.4 Marquer une classe avec une interface

On souhaite doter la classe `Marqueur` de la propriété suivante :
— Si cette classe implémente l'interface `Affichage`, présentée ci-dessous, alors la méthode `toString` affichera les coordonnées du point et le libellé ;
— Si cette classe n'implémente pas l'interface `Affichage` alors l'appel à la méthode `toString` retournera une chaîne vide.

```
interface Affichable {  }
```

Éléments de correction

La modification de la méthode `toString()` de `Marqueur` pour prendre en compte le test de l'implémentation de l'interface `Affichage`

```
class Marqueur implements Comparable<Marqueur> {
  private Point p;
  private String libellé;

  public String toString() {
    if (this instanceof Affichable) {
       return p.toString() + ", libellé : " + libelle;
    }
    else {
       return "";
    }
  }
}
```

Et un exemple d'utilisation :

```
interface Affichable { }

class MarqueurVisible extends Marqueur implements Affichable {
  public MarqueurVisible(double x, double y, String libelle) {
    super(x,y,libelle);
  }
}

class MarqueurInvisible extends Marqueur {
```

```
10    public MarqueurInvisible(double x, double y, String libelle) {
11      super(x,y,libelle);
12    }
13  }
14
15  public class TestMarqueurAffichable {
16      public static void main(String[] args) {
17          Marqueur[] marqueurs = {new MarqueurVisible(2,3,"A"),
18                                  new MarqueurInvisible(1,2,"B")};
19          for(Marqueur m : marqueurs) {
20              System.out.println("marqueur = " + m);
21          }
22      }
23      /* trace d'exécution :
24         > java TestMarqueurAffichable
25         marqueur = (2.0,3.0), libellé : A
26         marqueur =
27      */
28  }
29
```