Séquence 15

Interfaces

Le terme interface est utilisé en Java pour désigner deux choses qui n'ont rien à voir :

— Les composants du programme qui gèrent les échanges avec des éléments extérieur au programme (par exemple l'interface utilisateur, l'interface graphique, l'interface avec la base de données, etc).

— Une construction du langage qui permet de spécifier des fonctionnalités communes à plusieurs classes. C'est le sujet du présent chapitre.

1 Présentation d'un exemple

L'exemple que nous allons utiliser pour ce cours consiste en des formes géométrique dans un plan. On utilise les coordonnées cartésiennes (abcisse et ordonnée, x et y) pour spécifier ces formes.

```
1  public class Point{
2      protected double x, y;
3      public Point (double xi, double yi){
4          x = xi;
5          y = yi;
6      }
7      public double distance(Point p2){
8        return Math.sqrt((this.x-p2.x)*(this.x-p2.x) +
9            (this.y-p2.y) *(this.y-p2.y));
10     }
11 }
12 public class Cercle{
13     protected Point centre;
14     protected double rayon;
15     public Cercle(Point ctr, double r){
16         centre = ctr;
17         rayon = r;
18     }
19     public double surface(){
20         return Math.PI *rayon *rayon;
21     }
22 }
23 public class Rectangle{
24     protected Point basGauche;
25     protected double dimHor, dimVer;
```

```
26      public Rectangle(Point bg, double dh, double dv){
27          basGauche = bg;
28          dimHor = dh;
29          dimVer = dv;
30      }
31      public double surface(){
32          return dimHor *dimVer;
33      }
34  }
35  public class Triangle{
36      protected Point p1, p2, p3;
37      public Triangle(Point p1i, Point p2i, Point p3i){
38          p1 = p1i;
39          p2 = p2i;
40          p3 = p3i;
41      }
42      public double surface(){
43          double a = p1.distance(p2);
44          double b = p1.distance(p3);
45          double c = p2.distance(p3);
46          double demiper = (a+b+c)/2;
47          return Math.sqrt(demiper*(demiper-a)*(demiper-b)*(demiper-c));
48      }
49  }
```

Pour simplifier la définition d'un rectangle, nous avons supposé que les côtés du rectangle sont parallèles aux deux axes du plan.

On voit dans ce programme que les trois classes Cercle, Rectangle et Triangle proposent une méthode pour calculer leur surface. Dans les trois cas, cette méthode ne prend pas de paramètres et renvoie un double. Le code des trois méthodes est différent.

2 Les formes géométriques avec une interface

Il est possible de créer une interface pour regrouper ce que ces classes ont en commun. L'interface décrit un ensemble de méthodes. Les classes déclarent *implémenter* l'interface, c'est-à-dire fournir les méthodes spécifiées.

La définition de l'interface est la suivante :

```
public interface AvecSurface{
    double surface(); // Pas de corps de méthode
}
```

Une interface est généralement définie dans un fichier source (fichier .java) séparé. Ici, ce sera dans la fichier `AvecSurface.java`. Ce qui suit dans l'interface est *l'entête* de la méthode, c'est à dire ce qui dans une classe précède l'accolade qui débute les instructions. Avec quand même un élément manquant : on ne déclare pas les méthodes de l'interface `public`, `private` ou `protected`. Il ne faut aucun de ces mots-clés, mais la méthode devra être publique dans les classes qui implémentent l'interface.

Ensuite, pour que l'interface ait un intérêt, il faut qu'il y ait des classes qui l'implémentent. Dans notre exemple, les trois classes qui représentent des formes géométriques vont implémenter l'interface

alors que la classe `Point` ne va pas l'implémenter parce qu'il n'y a pas de méthode `distance` dans cette classe et parce qu'un point n'a pas de surface.

Il faut juste changer dans le code donné ci-dessus les trois entêtes de classes (lignes 12, 23 et 35) pour ajouter une clause `implements AvecSurface`.

```
public class Cercle implements AvecSurface{
  ...
}
public class Rectangle implements AvecSurface{
  ...
}
public class Triangle implements AvecSurface{
  ...
}
```

3 L'interface est un type Java

La définition de l'interface crée un nouveau type Java que l'on peut utiliser pour déclarer une variable, un paramètre, un attribut ou la valeur renvoyée par une méthode. Ce type contient les instances de toutes les classes qui l'implémentent.

Par exemple, dans une variable de type `AvecSurface`, je peux mettre une instance de `Cercle` parce que cette classe implémente `AvecSurface`. Je peux également mettre un rectangle ou un triangle dans cette variable.

Les objets cercles ont un type de plus. Ils ont le type `Cercle` par instanciation, le type `Object` par héritage et le type `AvecSurface` en plus.

Les seules opérations que l'on peut faire sur des variables qui ont pour type une interface sont l'affectation, les comparaisons == et != et l'appel des méthodes de l'interface.

```
public class Plan2{
    public static void main(String[] args){
        Point p1 = new Point(1,3);
        Point p2 = new Point(1,5);
        Point p3 = new Point(2,4);
        Triangle t = new Triangle(p1,p2,p3);
        Cercle c = new Cercle(p1,2.5);
        Rectangle r = new Rectangle(p1,2.7,5.0);
        AvecSurface as = t;
        AvecSurface[] tab = new AvecSurface[3];
        tab[0] = new Triangle(p1,p2,p3);
        tab[1] = new Rectangle(p1,2.7,5.0);
        tab[2] = new Cercle(p1,2.5);
        for (int i=0; i<3; i++){
            System.out.println("surface␣de␣tab[" + i + "]:␣" +
                                        tab[i].surface());
        }
        if ((tab[0].surface() > tab[1].surface())&&
            (tab[0].surface() > tab[2].surface())){
            System.out.println("tab[1]␣est␣le␣plus␣grand");
        }else{
            System.out.println("tab[2]␣est␣le␣plus␣grand");
```

```
        }
    }
}
```

On peut réaliser une conversion explicite pour passer d'un type interface à une des classes qui l'implémentent. Comme pour toute conversion explicite, il y a un risque de levée d'exception `ClassCastException` à l'exécution.

```
public class Plan3{
    public static void main(String[] args){
        Point p1 = new Point(1,3);
        AvecSurface forme = new Cercle(p1,15);
        Cercle cercle;
        Triangle triangle;
        cercle = (Cercle) forme;
        System.out.println("Première␣conversion␣réussie");
        triangle = (Triangle) forme;
        System.out.println("Second␣conversion␣réussie");
    }
}
```

À l'exécution :

```
> java Plan3
Première conversion réussie
Exception in thread "main" java.lang.ClassCastException:
class Cercle cannot be cast to class Triangle
        at Plan3.main(Plan3.java:9)
```

4 Programmation abstraite grâce aux interfaces

Supposons que nous n'ayons pas d'interface et que nous voulions comparer la surface de deux figures. Il faudrait dans chaque classe plusieurs méthodes pour comparer un objet de la classe avec un autre objet qui peut être de la même classe ou d'un autre genre de figure. Le code de ces différentes méthodes serait le même, mais le type du paramètre serait différent. Voyons le cas d'une méthode `compareSurface` qui renvoie un entier négatif si l'objet `this` a une surface plus petite, 0 si les deux surfaces sont égales et un entier positif sinon.

Cela donnerait le code suivant pour la classe `Cercle`.

```
public class Cercle{
    protected Point centre;
    protected double rayon;
    public Cercle(Point ctr, double r){
        centre = ctr;
        rayon = r;
    }
    public double surface(){
        return Math.PI *rayon *rayon;
    }
    public int compareSurface(Cercle c){
        int res=0;
        if (this.surface() < c.surface()){
```

```
            res = -1;
        }else if (this.surface() > c.surface()){
            res = 1;
        }
        return res;
    }
    public int compareSurface(Rectangle c){
        int res=0;
        if (this.surface() < c.surface()){
            res = -1;
        }else if (this.surface() > c.surface()){
            res = 1;
        }
        return res;
    }
    public int compareSurface(Triangle c){
        int res=0;
        if (this.surface() < c.surface()){
            res = -1;
        }else if (this.surface() > c.surface()){
            res = 1;
        }
        return res;
    }
}
```

Grâce au type `AvecSurface`, on peut écrire une seule méthode qui accepte en paramètre aussi bien un cercle qu'un rectangle ou un triangle. On peut ajouter cette méthode à l'interface AvecSurface.

```
interface AvecSurface{
    double surface(); // Pas de corps de méthode
    int compareSurface(AvecSurface as);
}
public class Cercle implements AvecSurface{
    protected Point centre;
    protected double rayon;
    public Cercle(Point ctr, double r){
        centre = ctr;
        rayon = r;
    }
    public double surface(){
        return Math.PI *rayon *rayon;
    }
    public int compareSurface(AvecSurface as){
        int res=0;
        if (this.surface() < as.surface()){
            res = -1;
        }else if (this.surface() > as.surface()){
            res = 1;
        }
        return res;
    }
}
```

On ajoute de la même façon la méthode `compareSurface` aux classes `Rectangle` et `Triangle` qui implémentent elles aussi cette interface. Le code de la méthode est identique dans les trois classes.

5 Interface et héritage

L'héritage a des répercussions sur les interfaces et leur implémentation.

Si une classe implémente une interface, alors toutes ses sous-classes l'implémentent aussi, même s'il n'y a pas de déclaration explicite.

```
1  import java.awt.Color;
2  public class CercleColore extends Cercle{
3      protected Color couleur;
4      public CercleColore(Point p, double r, Color c){
5          super(p,r);
6          couleur = c;
7      }
8  }
9  public class Plan6{
10     public static void main(String[] args){
11         Point centre = new Point(5,1);
12         CercleColore forme = new CercleColore(centre,5.0,Color.BLACK);
13         AvecSurface asf = forme;
14     }
15 }
```

Dans cet exemple, la classe `CercleColore` ne déclare pas qu'elle implémente `AvecSurface`. Mais elle l'implémente quand même. L'affectation opérée dans `Plan6.main` à la ligne 13 est correcte. Le programme se compile et s'exécute sans erreur.

Il est possible d'utiliser l'héritage sur les interfaces aussi. Une interface peut hériter d'une autre interface avec une clause `extends`. Dans ce cas, cette interface hérite des méthodes de sa super-interface (interface mère) et y ajoute ses propres méthodes.

```
import java.awt.Color;
public interface AvecSurfaceEtCouleur extends AvecSurface{
    Color getColor();
}
```

L'interface `AvecSurfaceEtCouleur` a trois méthodes : `surface`, `compareSurface` et `getColor`. Toute classe qui l'implémente doit comporter ces trois méthodes.

Une interface qui ne déclare aucune interface mère a quand même un super-type : il s'agit de `Object`.

6 Une classe peut implémenter plusieurs interfaces

Une classe peut implémenter plusieurs interfaces. Dans ce cas, la clause `implements` est suivie d'une liste de plusieurs interfaces séparées par une virgule.

Par exemple, la classe `String` implémente trois interfaces. Son entête est le suivant :

```
public class String implements Serializable, Comparable, CharSequence
```

Il y a donc dans la classe toutes les méthodes de ces trois interfaces.

7 Conclusion

Les interfaces sont un mécanismes de création de types qui est parfois en concurrence avec l'héritage. Nous verrons dans une autre séquence de cours comment on peut choisir le dispositif le mieux adapté à un besoin particulier.

Par ailleurs, les interfaces les plus intéressantes de la librairie Java font appel à la généricité. Il y a une sorte d'affinité entre interfaces et généricité. Nous verrons cela dans la séquence de cours consacrée à la généricité.