TD n° 1
Licence Informatique (L2)
« Programmation objet avancée »

F. BERTRAND

Année universitaire 2020–2021

Version avec éléments de correction

Concepts abordés :
- Méthodes de classe
- Principe d'encapsulation
- Utilisation de la relation d'héritage (spécialisation, généralisation)

1 Représentation de nombres complexes

Pour illustrer la notion de méthode de classe, nous allons définir une classe `Complexe` représentant un *nombre complexe* à l'aide d'une représentation cartésienne constituée d'une partie réelle et d'une partie imaginaire ($z = x + iy$).

Travail à effectuer :

1. Définir un ensemble de constructeurs permettant de créer un nombre complexe :
   - en précisant la valeur des parties réelle et imaginaire ;
   - en ne donnant aucun paramètre (les 2 parties sont alors égales à zéro) ;

Éléments de correction

```
1  public class Complexe
2  {
3      private double reel, img;
4
5      public Complexe(double x, double y)
6      {
7          this.reel = x;
8          this.img = y;
9      }
10
11     public Complexe()
12     {
13         this(0,0);
14     }
15 }
```

Noter l'appel d'un constructeur à partir d'un autre constructeur (appel `this(...)`) ce qui permet de réutiliser du code plutôt que de faire du copier-coller de code (ce qui n'est jamais une bonne chose).

2. Définir l'addition de 2 nombres complexes, à la fois comme une méthode d'instance et comme une méthode de classe. Le résultat retournera un nouveau nombre complexe correspondant à la somme des 2 nombres additionnés.

Éléments de correction

```
public class Complexe
{
    . . .

    public Complexe additionner(Complexe c)
    {
        return new Complexe(this.reel + c.reel, this.img + c.img);
    }

    public static Complexe additionner(Complexe c1, Complexe c2)
    {
        return new Complexe(c1.reel + c2.reel, c1.img + c2.img);
    }
}
```

Avec la méthode d'instance, l'addition n'apparaît qu'avec un seul nombre complexe en paramètre car l'autre nombre est représenté par l'objet sur lequel est appelée la méthode `additionner`*, exemple :*

```
Complexe resultat = c1.additionner(c2); // resultat = c1 + c2
```

3. Définir la multiplication de 2 nombres complexes sous la forme de deux méthodes d'instance. La première prenant un complexe en paramètre et la seconde prenant simplement un réel en paramètre. Une seule méthode (la première) est nécessaire, cependant la présence de la seconde fera que la multiplication d'un complexe avec un réel sera plus simple (pas de nécessité de créer un complexe ayant une partie imaginaire nulle) et plus efficace (moins de calculs).

Éléments de correction

```
public class Complexe
{
    . . .

    public Complexe multiplier(Complexe c)
    {
        double reel = this.reel * c.reel - this.img * c.img;
        double imag = this.reel * c.img + this.img * c.reel;
        return new Complexe(reel, imag);
    }

    public Complexe multiplier(double d)
    {
        return new Complexe(this.reel * d, this.img * d);
    }

}
```

La méthode `multiplier` *est dite* ***surchargée*** *car elle existe en deux versions et c'est lors de l'appel, grâce aux (types des) paramètres fournis, que le compilateur déterminera la méthode à appeler, exemple :*

```
Complexe resultat1 = c1.multiplier(c2); // multiplier(Complexe c) appelée
Complexe resultat2 = c1.multiplier(2.3); //multiplier(double d) appelée
```

La classe complète :

```
public class Complexe
{
    // la representation interne choisie
    // a definir privée !...
    private double reel, img;

    // les constructeurs
    public Complexe(double x, double y)
    {
        this.reel = x;
        this.img = y;
    }

    public Complexe()
    {
        this(0,0);
    }

    // l'addition avec une methode d'instance
    public Complexe additionner(Complexe c)
    {
        return new Complexe(this.reel + c.reel, this.img + c.img);
    }

    // l'addition avec une methode de classe
    public static Complexe additionner(Complexe c1, Complexe c2)
    {
        return new Complexe(c1.reel + c2.reel, c1.img + c2.img);
    }

    // versions surchargees pour la multiplication
    public Complexe multiplier(Complexe c)
    {
        double reel = this.reel * c.reel - this.img * c.img;
        double imag = this.reel * c.img + this.img * c.reel;
        return new Complexe(reel, imag);
    }

    public Complexe multiplier(double d)
    {
        return new Complexe(this.reel * d, this.img * d);
    }

    public String toString()
    {
        return this.reel + " + " + this.img + "i";
    }
}
```

et celle montrant son utilisation :

```
public class TestComplexe
{
    public static void main(String[] args)
    {
        Complexe c1 = new Complexe(2, 4);
        Complexe c2 = new Complexe(1, 3);
        Complexe resAdd = c1.additionner(c2);
        System.out.println(c1 + ".additionner(" + c2 + ") = " + resAdd);
        resAdd = Complexe.additionner(c1, c2);
        System.out.println("additionner(" + c1 + "," + c2 + ") = " + resAdd);
        Complexe resMult = c1.multiplier(c2);
```

```
        System.out.println("multiplier(" + c1 + "," + c2 + " = " + resMult);
        resMult = c1.multiplier(3);
        System.out.println("multiplier(" + c1 + "," + c2 + " = " + resMult);
    }
}
```

2 Figures géométriques

Le but de cet exercice est d'illustrer la notion d'héritage en utilisant une hiérarchie de figures géométriques. Soit les classes `Rectangle` et `Cercle` suivantes :

```
public class Rectangle {
        private double largeur;
        private double longueur;
        public Rectangle(double larg, double longueur) {
                this.largeur = larg;
                this.longueur = longueur;
        }
        public double donneSurface()  { return this.largeur * this.longueur; }
        public double donneLongueur() { return this.longueur; }
        public double donneLargeur()  { return this.largeur;  }
        public void changeLargeur(double l)  { this.largeur = l;  }
        public void changeLongueur(double l) { this.longueur = l; }
}
```

```
public class Cercle {
        private double x, y;  // abscisse et ordonnee du centre
        private double rayon;
        public Cercle(double x, double y, double r) {
                this.x = x;
                this.y = y;
                rayon = r;
        }
        public void affiche() {
                System.out.println("centre = (" + this.x + ", " + this.y + ")");
        }
        public double donneX() { return this.x; }
        public double donneY() { return this.y;        }
        public void changeCentre(double x, double y) {
                this.x = x;
                this.y = y;
        }
        public double donneSurface() { return Math.PI * this.rayon * this.rayon; }
        public boolean estInterieur(double x, double y) {
                return (((x - this.x) * (x - this.x) + (y - this.y) * (y - this.y))
                                                <= this.rayon * this.rayon);
        }
        public double donneRayon() { return this.rayon; }
        public void changeRayon(double r) {
                if (r < 0.0)
                   r = 0.0;
                this.rayon = r;
        }
}
```

Travail à réaliser :

1. Définir une classe `RectangleColoré` qui hérite de `Rectangle` et possède un attribut `couleur`.

Éléments de correction

```
class RectangleColore extends Rectangle {
    private int couleur;
    public RectangleColore(double larg, double longueur, int c) {
        super(larg, longueur);
        this.couleur = c;
    }
}
```

J'insiste ici sur la fait que les attributs de la super-classe doivent être initialisés avec le constructeur de la super-classe (en qualifiant systématiquement `private` *les attributs de la super-classe cela force à respecter cette règle).*

2. Définir une classe `Figure` contenant :
   — Deux attributs `x` et `y` qui représentent le centre de la figure ;
   — Un constructeur prenant les coordonnées du centre en paramètres.
   Faire ensuite hériter les classes `Cercle` et `Rectangle` de la classe `Figure`.

Éléments de correction

La classe `Figure` *:*

```
public class Figure {
    private double x;  // abscisse du centre
    private double y;  // ordonnee du centre
    public Figure(double x , double y){
        this.x = x;
        this.y = y;
    }
    // methodes accesseur
    public double donneX() {
        return this.x;
    }
    public double donneY() {
        return this.y;
    }
    public void changeCentre(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

Pour l'héritage de la classe `Figure`*, il faut spécifier les liens d'héritage aux deux classes concernées.*

```
class Rectangle extends Figure { ... }
class Cercle extends Figure { ... }
```

puis modifier les constructeurs de manière appropriée :

```
    //constructeur de la classe Cercle
    public Cercle(double x, double y, double r) {
        super(x,y);
        this.rayon=r;
    }
    ...
    //constructeur de la classe Rectangle
    public Rectangle(double x, double y,double larg, double longueur) {
        super(x,y);
        this.largeur = larg;
        this.longueur = longueur;
    }
```

```
13      ...
14      //constructeur de la classe RectangleColore
15      public RectangleColore(double x, double y,
16                             double larg, double longueur, int c) {
17          super(x,y,larg, longueur);
18          this.couleur = c;
19      }
```

3 Hiérarchie d'héritage

Le but de cet exercice est d'utiliser la relation d'héritage de manière ascendante (généralisation). À partir de l'ensemble des classes présentées sur la figure 1 concevoir une hiérarchie de classes permettant de de factoriser les définitions d'attributs.

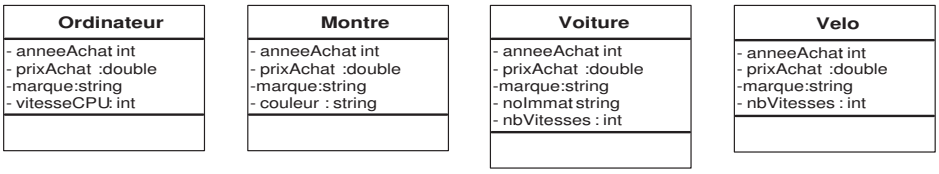

FIGURE 1 – Les classes à restructurer

Éléments de correction

Dans un premier temps, on peut adopter la hiérarchie de classes suivante :

```
1   class Bien {
2     private int anneeAchat;
3     private double prixAchat;
4     private String marque;
5   }
6
7   class Ordinateur extends Bien {
8     private int vitesseCpu;
9   }
10  class Montre extends Bien {
11    private String couleur;
12  }
13  class Voiture extends Bien {
14    private String noImmat;
15    private int nbVitesses;
16  }
17  class Velo extends Bien {
18    int nbVitesses;
19  }
```

Mais il est possible de factoriser un peu plus en créant une classe `Véhicule` *comprenant l'attribut* `nbVitesses`*...*

```
1   class Vehicule extends Bien {
2     private int nbVitesses;
3   }
4
5   class Voiture extends Vehicule {
6     private String noImmat;
7   }
8   class Velo extends Vehicule {
9   }
```

Puis on définit les constructeurs :

— *la première instruction doit être un appel à un constructeur de la super-classe (si des paramètres d'initialisation doivent lui être passés);*
— *sinon en l'absence d'appel à un constructeur spécifique de la super-classe, le compilateur génèrera automatiquement un appel au constructeur par défaut de la super-classe avec une erreur possible si le constructeur par défaut n'existe pas;*

Syntaxe pour appeler un constructeur de la super-classe : `super(...);`

```
1  public class Bien {
2    private int anneeAchat;
3    private double prixAchat;
4    private String marque;
5    public Bien (int a, double p, String m) {
6      this.anneeAchat = a;
7      this.prixAchat = p;
8      this.marque = m;
9    }
10 }
11 class Ordinateur extends Bien {
12   private int vitesseCpu;
13   public Ordinateur (int a, double p, String m, int v) {
14     super(a, p, m);
15     this.vitesseCpu = v;
16   }
17 }
18 class Montre extends Bien {
19   private String couleur;
20   public Montre (int a, double p, String m, String c) {
21     super(a, p, m);
22     this.couleur = c;
23   }
24 }
25
26 class Vehicule extends Bien {
27   private int nbVitesses;
28   public Vehicule (int a, double p, String m, int n) {
29     super(a, p, m);
30     nbVitesses = n;
31   }
32 }
33 class Voiture extends Vehicule {
34   private String noImmat;
35   public Voiture (int a, double p, String m, int n,
36             String noImmat) {
37     super(a, p, m, n);
38     this.noImmat = noImmat;
39   }
40 }
41 class Velo extends Vehicule {
42   public Velo (int a, double p, String m, int n) {
43     super(a, p, m, n);
44   }
45 }
```

Pour les constructeurs, pourquoi l'appel à `super` *est-il nécessaire ?*
— *principe de délégation de travail : le (ou les) constructeur(s) de chaque classe s'occupent de l'initialisation des attributs de leur classe;*
— *de plus si les attributs sont qualifiés* `private` *(ce qui est recommandé) alors il n'existe pas d'autres moyens de réaliser ces initialisations.*

Pour résumer :

1. *l'héritage entre classes permet de factoriser les déclarations d'attributs;*
2. *si le constructeur d'une sous-classe doit transmettre des paramètres à un constructeur de la super-classe alors il doit l'appeler à l'aide de* `super(...)` *en lui passant ces paramètres.*