Fiche 7 : classes abstraites, et retour sur la visibilité

IPO – Introduction à la programmation objet

Thibaut Balabonski @ Université Paris-Saclay
`https://www.lri.fr/~blsk/IPO/`

V1, automne 2022

Rappel de deux éléments des fiches précédentes.

- Le mécanisme d'*extension* permet de définir une nouvelle classe héritant du contenu d'une classe pré-existante. Avantage : on évite d'écrire plusieurs fois le même code.
- Une *interface* définit un type pouvant être réalisé par plusieurs classes concrètes. Avantage : on peut grouper des éléments de ces différentes classes concrètes dans une même structure, ou les faire traiter par une même méthode.

Les *classes abstraites* combinent les deux aspects précédents.

Classe abstraite Une classe abstraite est une classe dans laquelle certaines méthodes sont seulement *déclarées*, et pas définies. Une telle classe contient, comme une classe ordinaire : des déclarations d'attributs et des définitions de constructeurs et de méthodes. Elle peut contenir, en plus, des *méthodes abstraites*, c'est-à-dire de simples déclarations de méthodes similaires à celles présentes dans les interfaces. Une déclaration de méthode indique le type de retour, les types des arguments, et termine par un point-virgule sans code ni accolades. Chaque méthode ou classe asbtraite est accompagnée du mot-clé **`abstract`**.

```
abstract class Forme { // classe générale pour des formes géométriques
  private double cx, cy;  // coordonnées du centre de la forme
  public Forme(double cx, double cy) { this.cx = cx; this.cy = cy; }
  public void translation(double dx, double dy) { cx += dx; cy += dy; }

  public abstract void agrandissement(double f);  // agrandissement d'un facteur f
                                                  // le code dépend de la forme
}
```

Note : contrairement aux méthodes déclarées dans les interfaces, les méthodes abstraites ne sont pas systématiquement publiques. Il faut donc indiquer leur visibilité explicitement, comme pour les attributs et les méthodes concrètes.

Même si une classe abstraite définit un constructeur, il est impossible d'en construire une instance avec **`new`**. Si l'on tente de le faire, le compilateur Java déclenche une erreur avant même que l'on puisse tenter d'exécuter le programme.

```
Forme f = new Forme(1., 2.);
              ^^^^^
java: Forme is abstract; cannot be instantiated
```

En effet, l'existence d'un tel objet `f` serait problématique, puisqu'il n'aurait aucun code à exécuter au moment d'un appel `f.agrandissement(2.);`.

Concrétisation On concrétise une classe abstraite `A` en définissant une nouvelle classe `C` qui :

- étend la classe abstraite `A`,
- fournit des définitions pour les méthodes abstraites de `A`.

Il s'agit d'une extension comme les autres : tous les attributs et toutes les méthodes définies par `A` sont bien conservés. On ajoute seulement :

- les déclarations des attributs supplémentaires,
- les définitions des constructeurs,
- les définitions des méthodes abstraites,
- (et éventuellement les définitions de nouvelles méthodes non mentionnées dans `A`).

Les constructeurs de la classe concrète `C` peuvent faire appel aux constructeurs de la classe abstraite `A` via **`super`**.

```
class Cercle extends Forme { // un cas concret de forme
  private double r;  // rayon
  public Cercle(double cx, double cy, double r) {
    super(cx, cy);  // ne pas oublier d'initialiser les coordonnées du centre
    this.r = r;
  }
  public void agrandissement(double f) { this.r *= f; }  // ici, on sait quoi faire
}
```

Comme avec les interfaces, on peut ensuite créer un objet ayant le type de la classe abstraite à l'aide du constructeur de la classe concrète.

```
Forme f = new Cercle(1., 2., 1.);
```

Avantage par rapport aux interfaces : le code de la classe abstraite est écrit une fois pour toutes, et partagé par toutes les classes concrètes. On peut ainsi multiplier les formes, en n'écrivant que la partie du code qui est spécifique à chacune.

```
class Rectangle extends Forme { // un autre cas concret
  private double h, l;  // hauteur, largeur
  public Rectangle(double cx, double cy, double h, double l) {
    super(cx, cy); this.h = h; this.l = l;
  }
  public void agrandissement(double f) { this.h *= f; this.l *= f; }
}
```

Les instances de différentes classes concrètes peuvent ainsi être regroupées au sein d'un même ensemble de formes.

```
ArrayList<Forme> arr = new ArrayList<>();
arr.add(new Cercle(1., 2., 1.));
arr.add(new Rectangle(0., 1., 2., 2.));
```

Toute classe possédant une méthode abstraite doit elle-même être déclarée abstraite, sans quoi le compilateur Java produit une erreur.

```
class Forme {
^
Class 'Forme' must either be declared abstract or implement abstract method
'agrandissement' in 'Forme'
```

Cela vaut y compris pour une classe `C` étendant une classe abstraite `A`, mais sans fournir de définitions pour certaines des méthodes abstraites de `A`.

```
abstract class Inclinable extends Forme {
  private double angle;
  public Inclinable(double cx, double cy, double a) {
    super(cx, cy); this.angle = a;
  }
  public void rotation(double a) { this.angle += a; }
  // pas de définition pour agrandissement
}
class Rectangle extends Inclinable { /* code déjà vu + constructeur à adapter */ }
```

Techniquement, rien n'interdit de déclarer **`abstract`** une classe dont toutes les méthodes seraient définies. En revanche, cela empêcherait d'utiliser la construction d'une instance avec **`new`**.

Retour sur les visibilités Les différents membres d'une classe (attributs, constructeurs, méthodes) sont toujours associés à une *visibilité*, qui spécifie *qui* peut accéder à ce membre ou l'utiliser. On a déjà vu :

- **`public`**, souvent utilisé pour les méthodes et constructeurs, qui ne donne aucune restriction d'accès,
- **`private`**, souvent utilisé pour les attributs, qui ne permet l'accès à un membre donné que dans le code de la classe elle-même.

Il existe deux niveaux de visibilité supplémentaires, intermédiaires entre **`public`** et **`private`**.

- Sans indication de visibilité on se trouve dans le mode par défaut, nommé *package-private*, qui rend le membre visible dans l'ensemble du *package* où est définie la classe (en gros : le répertoire contenant le fichier, on reviendra sur cette notion).
- Avec l'indication **`protected`**, le membre est visible dans l'ensemble du *package*, ainsi que dans toutes les sous-classes, directes ou indirectes.

Ces niveaux intermédiaires sont légitimement utilisables lorsque l'on développe plusieurs classes en forte interaction.

```
abstract class Forme {
  protected double cx, cy;  // les formes concrètes peuvent avoir besoin
                            // de connaître les coordonnées du centre
  ...
}

class Cercle extends Forme {
  ...
  public boolean contains(x, y) {
    return (x-cx)*(x-cx) + (y-cy)*(y-cy) <= r*r;  // en effet
  }
}
```

Constantes Lors de l'introduction d'une variable en Java, on peut déclarer que la valeur de celle-ci n'est pas modifiable avec le mot-clé **`final`**. Avec une telle déclaration, on peut réaliser une unique affectation à la variable (bonne pratique : la coupler à la déclaration). Les éventuelles affectations suivantes sont rejetées par le compilateur.

```
final int x = 1;
x = 2;
^
Variable 'x' might already have been assigned to
```

Cette qualification s'applique de même aux attributs des objets. Dans ce cas, l'affectation initiale peut être soit couplée à la déclaration, soit réalisée par le constructeur.

```
class Point {
  public final double x, y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
}
```

Toute tentative d'affectation ailleurs que dans le constructeur sera rejetée, car toute méthode autre que le constructeur est susceptible d'être appelée plusieurs fois dans la vie d'un objet.

```
  public void setX(double x) { this.x = x; }
                               ^^^^^^
  Cannot assign a value to final variable 'x'
```

Le compilateur fait des vérifications assez précises : un attribut **`final`** doit être initialisé, soit à sa déclaration, soit *dans chaque constructeur*, mais pas à ces deux endroits à la fois.

Note : un attribut **`final`** étant immuable, on élimine une des raisons poussant à déclarer les attributs **`private`** (personne ne pourra briser par inadvertance les invariants d'une classe en modifiant cet attribut, puisque la modification est impossible). Sauf à vouloir empêcher les utilisateurs de tenir compte de la valeur d'un tel attribut, on peut donc le rendre **`public`**, comme dans l'exemple ci-dessus. Fonctionnellement, cela équivaut à avoir des *getters* mais pas de *setters*, mais sans besoin d'alourdir le code avec des appels de méthode pour accéder à ces valeurs.

```
Point p = new Point(1., 2.);
double n2 = p.x * p.x + p.y * p.y;
```

Dans le cas d'un attribut **`static`**, qui est lié à la classe elle-même et pas à une instance produite par le constructeur, la seule possibilité est d'initialiser à la déclaration.

```
public static final int REPONSE = 42;
```

Ce dernier exemple est une bonne manière d'introduire une constante globale dans un programme.

Le mot-clé **`final`** s'applique aussi à des méthodes et à des classes, pour contrôler les extensions et redéfinitions :

— une méthode **`final`** ne peut pas être redéfinie dans une sous-classe,
— une classe **`final`** ne peut pas être étendue.