Chapitre 9

Classes abstraites et interfaces

1 Classes et méthodes abstraites

On a vu que l'héritage permettait de regrouper dans un tableau d'un type donné des objets de ce type mais aussi des objets d'un type dérivé, par exemple Horaire et HorairePrecis.

Changeons d'exemple et revenons à des formes géométriques. Nous en avons étudié deux : le rectangle et le cercle. On peut facilement en imaginer d'autres (triangle, ellipse, etc.) Toutes posséderaient une méthode aire() et une méthode perimetre().

Si l'on voulait stocker toutes ces formes dans un tableau de formes, il serait pratique qu'elles dérivent toutes d'une même super classe : la classe Forme. Et comme chaque forme propose les méthodes aire() et perimetre(), il serait naturel d'en doter la classe Forme elle même.

Le problème est qu'on ne sait pas implanter ces méthodes dans le cas général : le calcul de la surface d'un carré n'est pas le même que celui de la surface d'un cercle.

Question. Quel devrait être le code des méthodes aire() et perimetre() de la classe générique Forme ?

Réponse. Aucun !

La programmation par objets permet de définir une méthode sans l'implanter : on la déclare avec le modificateur *abstraite* (*abstract* en java). La définition d'un méthode abstraite se limite à sa signature.

Toute sous-classe voulant hériter de Forme a maintenant pour obligation de fournir une implantation concrète des méthodes aire() et perimetre() pour pouvoir être instanciée.

Ainsi, rédiger une classe abstraite sert essentiellement à faire du typage, en indiquant de manière abstraite ce que toute sous-classe est censée réaliser concrètement.

1.1 Règles des classes et méthodes abstraites

Les règles suivantes s'appliquent :
- Toute classe contenant une méthode abstraite est elle-même obligatoirement abstraite, et doit être déclarée abstraite.
- Une classe abstraite ne peut pas être instanciée.
- Une sous-classe d'une classe abstraite ne peut être instanciée que si elle redéfinit chaque méthode abstraite de sa super-classe en fournissant une implantation. Une telle classe est souvent appelée une sous-classe *concrète*.
- Si une sous-classe d'une classe abstraite n'implante pas *toutes* les méthodes abstraites dont elle hérite, alors cette sous-classe est elle-même abstraite.
- Les méthodes déclarées *statiques*, *privées* ou *finales* n'ont pas le droit d'être abstraites.
- Une classe déclarée *finale* n'a pas le droit de contenir de méthode abstraite.
- Une classe peut être déclarée abstraite même si elle ne possède pas de méthode déclarée abstraite. C'est le cas quand les méthodes ne fournissent qu'une implantation partielle qui sera complétée dans chaque sous-classe.

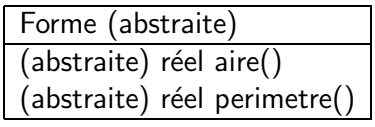
À retenir absolument.
- Les classes abstraites ne peuvent pas être instanciées.
- Hériter (concrètement) d'une classe abstraite donne l'obligation d'implanter chaque méthode abstraite de la super-classe.

1.2 Utilisation des classes abstraites

Les classes abstraites sont faites pour typer, et être héritées en sous-classe concrètes.

Soit la classe Forme.

Son modèle est :



Son code Java est :

```
public abstract class Forme {
    public abstract double aire();
    public abstract double perimetre();
}
```

Les deux classes Rectangle et Cercle que nous avons traitées auraient pu être définies par héritage de la classe abstraite Forme :

public class Rectangle extends Forme
public class Cercle extends Forme

Observons maintenant l'extrait de code suivant :

```
Tableau de Forme tabForme
tabForme ← créer Forme[10]
tabForme[0] ← créer Cercle(5)
tabForme[1] ← créer Rectangle(3,4)
afficher(tabForme[0].aire())
afficher(tabForme[1].aire())
```

Ce code peut compiler sans problèmes car, bien que tabForme[0] et tabForme[1] soient devenues syntaxiquement des instances de Forme, elles possèdent bien une méthode aire() puisque celle-ci apparaît comme une méthode de la classe abstraite Forme.

Question. Qu'afficheraient les deux dernières lignes ?

Réponse. $78,54$ (l'aire du cercle de rayon 5), puis 12 (l'aire du rectangle de dimension 3×4). « Comme par magie », la méthode aire() invoquée a bien été celle de l'objet réel, bien que celui-ci soit vu dans les deux cas comme une instance de sa super-classe Forme. Ce mécanisme est connu sous le nom de liaison dynamique et permet d'assurer le polymorphisme, qui sera étudié au chapitre 11.

2 Interfaces

Nous avons étudié aussi des versions des classes Cercle et Rectangle qui étaient *positionnées* dans le plan par rapport à un point de référence : le centre pour le cercle, et le coin supérieur gauche pour le rectangle. On pourrait imaginer une classe abstraite nommée ObjetDansLePlan qui proposerait la méthode abstraite setPosition(Point p). Ainsi, des classes telles que Cercle et Rectangle pourraient être définies par héritage de ObjetDansLePlan.

Mais comme des formes positionnées dans le plan restent des formes, on voudrait pouvoir hériter aussi de Forme.

En Java, cela n'est pas possible car l'héritage multiple est interdit. La déclaration

```
public class Cercle extends Forme, ObjetDansLePlan { ...
```

serait rejetée à la compilation.

On va résoudre notre problème en utilisant le concept *d'interface*, que l'on va *implanter*. On ne peut hériter que d'une seule classe, mais on peut implanter un nombre quelconque d'interfaces.

2.1 Une interface définit une propriété transverse de différents objets

Voyons notre exemple différemment. Un cercle et un rectangle *sont des* formes, et il est naturel pour eux d'hériter de Forme. Le fait de pouvoir les positionner dans le plan relève

plus de la propriété d'être *positionnable* ou pas. Cette propriété pourrait concerner d'autres objets que des formes sur un écran graphique : images, zones de texte, icônes, etc. De plus, être positionnable ne concerne pas forcément toutes les formes géométriques : celles pour lesquelles on n'a pas défini un point de référence ne sont pas positionnables.

Une telle propriété, transverse de plusieurs classes d'objets, se définit par une interface.

Définition 9.1 (Interface)**.** *Une interface est un ensemble d'en-têtes de méthodes constituant une API, et assurant aux objets qui l'implantent la faculté d'être manipulables conformément à la propriété qu'ils annoncent posséder.*

Pour notre exemple, un objet positionnable se devra d'implanter une méthode nommée par exemple setPosition(), permettant de définir sa position. L'interface Positionnable se contentera de définir l'en-tête de cette méthode.

2.2 Définition d'une interface

La définition d'une interface ressemble beaucoup à celle d'une classe abstraite, mais le mot interface remplace les mots abstract et class (ou abstraite et classe en PDL++).

Une interface constitue, comme une classe ou une classe abstraite, une définition de type. Elle a pour but de définir l'interface, ou encore l'API (*Application Programming Interface*) d'une famille d'objets, sans définir aucune implantation de cette API.

Les règles suivantes s'appliquent :

- Une interface ne contient aucune implantation. Un point-virgule ';' remplace le corps de la méthode.
- Toutes les méthodes d'une interface sont implicitement abstraites, même si le modificateur abstract (ou abstraite) est omis.
- Toutes les méthodes d'une interface doivent être d'instance (⇒ pas de méthode statique dans une interface).
- Toutes les méthodes d'une interface sont implicitement publiques, même si le modificateur public est omis.
- Une interface n'a pas le droit de définir d'attributs, à l'exception de constantes déclarées à la fois static et final.
- Une interface n'a pas de constructeur.

2.3 Utilisation des interfaces

On n'hérite pas d'une interface mais on dit qu'on *l'implante*. On l'indique en java par le mot-clé implements (implante en PDL++) dans la définition de la classe.

Exemple.

```
public class Rectangle extends Forme implements ObjetDansLePlan { ...
```

Contrairement aux classes abstraites, les interfaces ne définissent aucune implantation, même partielle. Mais il est possible à une classe d'implanter plusieurs interfaces à la fois, alors qu'on n'a le droit d'hériter que d'une seule super-classe. Les différentes interfaces implantées sont séparées par des virgules.

En déclarant qu'on implante une interface, on s'oblige à fournir une implantation de *toutes* les méthodes de l'interface. Dans le cas contraire, une erreur est signalée à la compilation.

L'utilité est de pouvoir faire savoir aux programmeurs qui vont utiliser l'objet qu'on définit, que l'on se rend disponible pour telle ou telle opération définie dans l'interface. Ainsi, un programme peut manipuler un objet qu'il ne connaît pas avec les méthodes de l'interface, dès lors qu'il sait que l'objet implante cette interface. On peut donc faire des programmes génériques, qui manipulent des objets que l'on peut définir plus tard.

En résumé, en choisissant d'implanter une interface, on rend l'objet qu'on définit utilisable par des algorithmes existants.

Par exemple, une interface fournie par l'API Java est nommée Comparable. Elle impose une unique méthode nommée compareTo(...), qui établit une relation d'ordre entre les objets de la classe qui l'implantera. Si on souhaite définir une nouvelle classe, et qu'une collection d'objets (par exemple un tableau) de cette classe puisse être triée rien qu'en lui appliquant la méthode java prédéfinie sort(), on choisira d'implanter l'interface Comparable. Cela nous rend disponible pour les méthodes de tri de Java, qui manipulent les objets à trier à travers leur méthode compareTo(...).

2.4 Rédiger et nommer une interface

Puisqu'une classe implantant une interface a l'obligation d'implanter toutes les méthodes de l'interface pour pouvoir être instanciée, une interface contient en général un petit nombre de méthodes.

Le nom de l'interface doit refléter au mieux la propriété qu'elle traduit. On peut choisir un nom en « able » quand c'est possible. En effet, une interface définit le plus souvent un ensemble d'opérations applic**ables**, et le nom de l'interface traduit cette disponibilité pour ces opérations. Mais ce n'est pas toujours possible et on nomme alors l'interface comme une autre classe.

Exemples en java : Comparable, Cloneable, Iterable, Throwable, etc., mais aussi Comparator, Action, Set, ...

2.5 Un exemple complet (en Java)

Pour mieux nommer l'interface, on choisit de renommer ObjetDansLePlan par Positionnable.

```
public interface Positionnable {
    public void setPosition(Point p);
}
```

```
public class Rectangle extends Forme implements Positionnable {
    private double largeur;
    private double hauteur;
    private Point posCSG;

    // Constructeurs
    public Rectangle(double largeur, double hauteur, Point p) {
        this.largeur = largeur;
        this.hauteur = hauteur;
        this.posCSG = p;
    }

    public Rectangle(double largeur, double hauteur) {
        this(largeur, hauteur, new Point(0,0));
    }

    public Rectangle() {
        this(1,1, new Point(0,1));
    }

    // Méthodes

    // méthodes abstraites de Forme
    public double aire() {
        return largeur*hauteur;
    }

    public double perimetre() {
        return 2*(largeur+hauteur);
    }

    // méthodes de l'interface FormePositionnable
    public void setPosition(Point p) {
        posCSG = p;
    }
}
```