Chapitre 7

Encapsulation et masquage des données

Rappel. Les données qui caractérisent l'état d'un objet sont enregistrés dans ses attributs, tandis que les manipulations possibles de l'objet sont fournies par ses méthodes.

1 Encapsulation

La programmation par objets réunit au sein d'une même structure, celle de classe, les données et les méthodes qui les manipulent.

Un avantage fondamental de cette façon de programmer est de pouvoir contrôler l'accès aux données : en ne manipulant un objet qu'au travers de ses méthodes, on a la garantie de conserver l'objet dans un état cohérent.

Une bonne pratique en POO est de ne pas accéder (lecture/écriture) directement aux attributs mais de passer par les méthodes.

Définition 7.1 (Encapsulation)**.** *L'encapsulation consiste à masquer les données au sein d'une classe, en ne les mettant à disposition que par l'intermédiaire des méthodes.*

Un objet est alors semblable à une capsule dont l'intérieur (les attributs) est caché. On ne peut agir sur la capsule qu'au travers d'un certain nombre de « leviers de commande ». Ces leviers sont les méthodes de l'objet.

Un autre avantage conféré par l'encapsulation est que, vu de l'extérieur, un objet sera vu comme une *boîte noire* ayant certaines propriétés et un comportement spécifié. La façon d'implanter ces propriétés est cachée aux utilisateurs de la classe. On peut changer l'implantation sans changer le comportement extérieur de l'objet. Cela permet donc de séparer la spécification du comportement d'un objet, de l'implantation pratique de cette spécification.

```
class Horaire {

 int h, m, s ; // les heures, minutes et secondes

 void setHoraire(int h, int m, int s) {
    if (h >= 0 && h < 23 && m >= 0 && m < 59 && s >= 0 && s < 59) {
        this.h = h ;
        this.m = m ;
        this.s = s ;
    }
 }
}
```

FIG. 7.1 – Une classe Horaire en Java

Prenons l'exemple d'une classe Horaire, tel que décrite dans la figure 7.1.

Il semble naturel de n'autoriser à changer l'horaire qu'au travers de la méthode setHoraire(...). En effet, celle-ci *protège* l'horaire, en ne le modifiant que pour un horaire correct. Soit $h1$ une instance de Horaire. Une tentative de mettre $h1$ à l'horaire 28h154m-12s par un appel à

h1.setHoraire(28,154,-12)

ne modifie pas l'horaire de $h1$ car la méthode setHoraire(...) « rejette » cet horaire inepte.

Telle qu'est écrite la classe Horaire, rien n'empêche un programmeur (mal intentionné ou distrait) de mettre $h1$ dans cet état incohérent par les instructions

h1.h = 28 ;
h1.m = 154 ;
h1.s = -12 ;

Comment forcer un utilisateur à passer par la méthode setHoraire() *?*

2 Masquage des données

Le masquage des données ou leur libre accès est obtenu au moyen de *modificateurs de visibilité*.

2.1 Le modificateur de visibilité private

On emploiera le mot clé privé en PDL++ devant la déclaration d'un attribut ou d'une méthode pour réaliser le masquage des données. En java, on utilise private. Un attribut (ou

une méthode) déclaré(e) private est invisible, et donc inaccessible, depuis l'extérieur de la classe.

```
class HorairePrivate {

 private int h, m, s ; // les heures, minutes et secondes, privees

 void setHoraire(int h, int m, int s) {
    if (h >= 0 && h < 23 && m >= 0 && m < 59 && s >= 0 && s < 59) {
        this.h = h ;
        this.m = m ;
        this.s = s ;
    }
 }
}
```

FIG. 7.2 – Protection des attributs h, m et s de la classe Horaire en Java

La figure 7.2 présente une version modifiée de la classe Horaire, dans laquelle les attributs h, m et s ont été masqués avec private. Ils ne sont alors accessibles que depuis l'intérieur même de la déclaration de la classe.

Ainsi, setHoraire(...) a toujours le droit de manipuler directement les attributs h, m et s, mais toute tentative d'accès direct à un attribut déclaré private depuis *l'extérieur* de la classe est rejeté à la compilation.

Soit $h2$ une instance de la classe Horaire ainsi modifiée. Les instructions
h2.h = 28 ;
h2.m = 154 ;
h2.s = -12 ;
sont rejetées à la compilation, car les attributs h, m et s sont masqués.

Des méthodes privées ? Le même mécanisme s'applique aux méthodes. Si une méthode est déclarée private, alors elle est masquée. Elle ne pourra pas être invoquée depuis l'extérieur de la classe. Seules les autres méthodes de la même classe sont autorisées à invoquer la méthode privée. L'utilité d'une méthode privée peut être de rendre un code plus clair, ou plus efficace, ou plus propre... Pour autant, une telle méthode n'a parfois pas sa place dans l'API de l'objet, et doit donc être cachée pour l'extérieur.

2.2 Le modificateur de visibilité public

Il existe un modificateur de visibilité dont l'effet est inverse à celui de private, défini par le mot clé public en PDL++ et en Java. Un attribut ou une méthode public est visible depuis

tout endroit d'un programme.

2.3 D'autres modificateurs de visibilité

Il existe un autre modificateur de visibilité : protected (protégé en PDL++). Il sera vu au chapitre suivant sur l'héritage.

Quand on ne précise aucun modificateur de visibilité, le modificateur par défaut n'est ni public, ni private ni protected. Il est nommé friendly en C++ mais n'a pas de nom en Java. Il est lié à la notion de *package* que nous n'étudions pas dans ce cours, et nous l'assimilerons à public.

3 *Getters* et *Setters*

3.1 Les *Getters* (ou accesseurs)

Les méthodes qui permettent de lire les valeurs des attributs depuis l'extérieur de la classe sont appelées des *getters*. Il est d'usage de les nommer en les préfixant par get. Ils sont généralement publics.

On aurait pu doter la classe Horaire de trois getters :

```
int getHeure() {
        return h;
}

int getMinute() {
        return m;
}

int getSeconde() {
        return s;
}
```

N.B. Quand la valeur renvoyée est un booléen, on utilise le préfixe is au lieu de get.

3.2 Les *Setters* (ou mutateurs)

Les méthodes qui permettent de modifier les valeurs des attributs depuis l'extérieur de la classe sont appelées des *setters*. Il est d'usage de les nommer en les préfixant par set. Ils sont généralement publics.

La classe Horaire possède déjà un setter : setHoraire(...).

Remarque. Si un objet est *non mutable*, on ne définit pas de setter dans sa classe. Un objet non mutable est un objet dont ont définit l'état (= la valeur des attributs) à l'instanciation, ces valeurs n'étant plus modifiées ensuite. Exemple : les objets Point ou NombreComplexe sont non mutables.

3.3 Les getters et setters contredisent-ils le masquage des données ?

La réponse est non !

D'une part, il n'est pas obligatoire de définir des getters et des setters pour des données que l'on veut garder secrètes. D'autre part, le fait de mettre à disposition des méthodes publiques afin de lire et d'écrire un attribut n'équivaut pas à rendre l'attribut lui-même public. Une différence cruciale est que les setters peuvent effectuer un contrôle des valeurs affectées.

Nous aurons dorénavant comme politique de déclarer tous les attributs privés, sauf si on sait donner un motif précis et bien identifié pour ne pas le faire (cas des constantes, par exemple).