Chapitre 5

Héritage

Une des principales qualités constitutives des langages à objets est la *réutilisabilité*, c'est-à-dire la réutilisation de classes existantes. La réutilisation de composants logiciels déjà programmés et fiables permet une économie de temps et d'erreurs dans la construction de nouveaux programmes. Chaque année, des millions de lignes de code sont écrites et seul un faible pourcentage est original. Si cela se conçoit dans un cadre pédagogique, ça l'est beaucoup moins dans un environnement industriel. Des algorithmes classiques sont reprogrammés des milliers de fois, avec les risques d'erreur que cela comporte, au lieu de faire partie de bibliothèques afin d'être mis à la disposition des programmeurs. La notion d'héritage, que nous allons aborder dans ce chapitre, est l'outil qui facilitera la conception des applications par réutilisation.

1 CLASSES HÉRITIÈRES

Dans le chapitre 7, nous avons défini une classe pour représenter et manipuler des rectangles. Si, maintenant, nous désirons représenter des carrés, il nous faut définir une nouvelle classe. Chaque carré est caractérisé par la longueur de son côté, son périmètre, sa surface, *etc*. La classe pour représenter les carrés est définie comme suit :

```
classe Carré
   {Invariant de classe : côté ⩾ 0}
   public périmètre, surface, côté

   {l'attribut}
   côté type réel
```

```
    {le constructeur}
    constructeur Carré(donnée c : réel)
        côté ← c
    fincons

    {les méthodes}
    fonction périmètre() : réel
    {Rôle: retourne le périmètre du carré}
        rendre 4×côté
    finfonc {périmètre}

    fonction surface() : réel
    {Rôle: retourne la surface du carré}
        rendre côté×côté
    finfonc {surface}
finclasse Carré
```

Vous pouvez constater que cette classe ressemble fortement à la classe `Rectangle`. Ceci est normal dans la mesure où un carré est un rectangle particulier dont la largeur est égale à la longueur. Aussi, plutôt que de définir entièrement une nouvelle classe, il est légitime de *réutiliser* certaines des caractéristiques d'un rectangle pour définir un carré. L'*héritage* est une relation entre deux classes qui permet à une classe de réutiliser les caractéristiques d'une autre. Nous définirons la classe `Carré` comme suit :

```
classe Carré hérite de Rectangle
    {Invariant de classe : longueur = largeur ⩾ 0}
    {le constructeur}
    constructeur Carré(donnée c : réel)
           Rectangle(c,c)
    fincons
finclasse Carré
```

Tous les attributs et toutes les méthodes de la classe `Rectangle`, la classe *parent*, sont accessibles depuis la classe `Carré`, le *descendant*. On dit que la classe `Carré` *hérite*[1] de la classe `Rectangle`. La classe `Carré` ne définit que son constructeur, qui appelle celui de sa classe parent, et hérite automatiquement des attributs, `largeur` et `longueur`, ainsi que des méthodes `périmètre` et `surface`.

La figure montre de façon graphique la relation d'héritage entre les classes `Rectangle` et `Carré`. Chaque classe est représentée par une boîte qui porte son nom et l'orientation de la flèche signifie *hérite de*.

La déclaration d'un carré `c` et le calcul de sa surface sont alors obtenus par :

```
variable c type Carré créer Carré(5)

... c.surface() ...
```

La réutilisabilité des classes déjà fabriquées est un intérêt majeur de l'héritage. Il évite une perte de temps dans la récriture de code déjà existant, et évite ainsi l'introduction de nouvelles

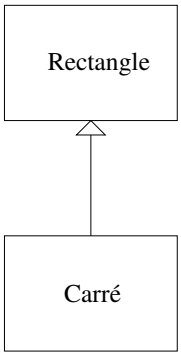

FIG. Relation d'héritage entre les classes `Rectangle` et `Carré`.

erreurs dans les programmes. L'héritage peut être vu comme un procédé de *factorisation*, par la mise en commun des caractéristiques communes des classes.

Comme pour une généalogie humaine, une classe héritière peut avoir ses propres descendants, créant ainsi un véritable arbre généalogique. La relation d'héritage est une relation transitive : si une classe B hérite d'une classe A, et si une classe C hérite de la classe B, alors la classe C hérite également de A. Dans beaucoup de langages de classe, toutes les classes possèdent un ancêtre commun, une classe souvent appelée *Object*, qui est la racine de l'arborescence d'héritage (voir la figure 2).

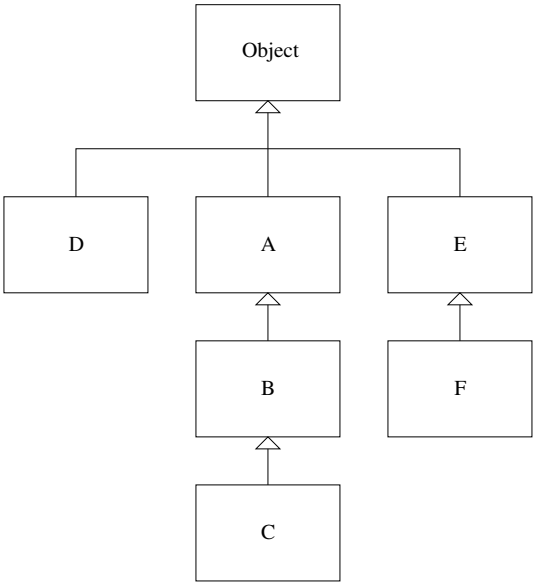

FIG.2 Un ancêtre commun : la classe Object.

Évidemment, les classes héritières peuvent définir leurs propres caractéristiques. La classe `Carré` possède déjà son propre constructeur, mais pourra définir, par exemple, une méthode de mise à jour du côté d'un carré.

```
classe Carré hérite de Rectangle
   {Invariant de classe : longueur = largeur ⩾ 0}
   public changerCôté
   {le constructeur}
   constructeur Carré(donnée c : réel)
         Rectangle(c,c)
   fincons

   procédure changerCôté(donnée c : réel)
   {Rôle: met la valeur du côté à la valeur c}
      changerLargeur(c)
      changerLongueur(c)
   finproc
finclasse Carré

...
c.changerCôté(10)
```

La relation d'héritage peut donc aussi être considérée comme un mécanisme d'*extension* de classes existantes, mais également de *spécialisation*. Les informations les plus générales sont mises en commun dans des classes parentes. Les classes se spécialisent par l'ajout de nouvelles fonctionnalités. Il est important de comprendre que n'importe quelle classe ne peut étendre ou spécialiser n'importe quelle autre classe. La classe `Carré` qui hériterait d'une classe `Individu` n'aurait aucun sens. Le mécanisme d'héritage permet de conserver une cohérence entre les classes ainsi mises en relation.

2 REDÉFINITION DE MÉTHODES

Lorsqu'une classe héritière désire modifier l'implémentation d'une méthode d'une classe parent, il lui suffit de redéfinir cette méthode. La *redéfinition* d'une méthode est nécessaire si on désire adapter son action à des besoins spécifiques. Imaginons, par exemple, que la classe `Rectangle` possède une méthode d'affichage, la classe `Carré` peut redéfinir cette méthode pour l'adapter à ses besoins.

```
classe Rectangle
   ...
   procédure afficher()
   {Rôle: affiche une description du rectangle courant}
       écrire("rectangle de largeur", largeur,
                      "et de longueur" , longueur)
   finproc
  ...
finclasse Rectangle

classe Carré hérite de Rectangle
   ...
   procédure afficher()
```

```
    {Rôle: affiche une description du carré courant}
        écrire("carré de côté égal à" , largeur)
    finproc
    ...
finclasse Carré
```

Dans le fragment de code suivant :

```
variable c type Carré créer Carré(5)
variable r type Rectangle créer Rectangle(2,4)

r.afficher()
c.afficher()
```

il est clair que c'est la méthode `afficher` de la classe `Rectangle` qui s'applique à l'objet `r` et celle de la classe `Carré` qui s'applique à l'objet `c`. Notez que les redéfinitions permettent de changer la mise en œuvre des actions, tout en préservant leur sémantique. Ainsi, la méthode `afficher` de la classe `Carré` ne devra pas calculer, par exemple, la surface d'un carré.

3 RECHERCHE D'UN ATTRIBUT OU D'UNE MÉTHODE

Si la méthode que l'on désire appliquer à une occurrence d'un objet d'une classe C n'est pas présente dans la classe, celle-ci devra appartenir à l'un de ses ancêtres.

D'une façon générale, chaque fois que l'on désire accéder à un attribut ou une méthode d'une occurrence d'objet d'une classe C, il (ou elle) devra être défini(e), soit dans la classe C, soit dans l'un de ses ancêtres. Si l'attribut ou la méthode n'appartient pas aux classes parentes, l'attribut ou la méthode n'est pas trouvé et c'est une erreur de programmation.

S'il y a eu des redéfinitions de la méthode, sa première apparition en remontant l'arborescence d'héritage est celle qui sera choisie. Ainsi, avec les déclarations suivantes :

```
variable c type Carré
variable r type Rectangle
```

`c.périmètre()`, provoque l'exécution de la méthode `périmètre` définie dans la classe `Rectangle`, alors que `r.changerCôté()` provoque une erreur.

4 POLYMORPHISME ET LIAISON DYNAMIQUE

Dans certains langages de programmation, une variable peut désigner, à tout moment au cours de l'exécution d'un programme, des valeurs de n'importe quel type. De tels langages sont dits *non typés* ou encore *polymorphiques*[2]. En revanche, les langages dans lesquels une variable ne peut désigner qu'un seul type de valeur sont dits *typés* ou *monomorphiques*. Ces derniers offrent plus de sécurité dans la construction des programmes dans la mesure où les

vérifications de cohérence de type sont faites dès la compilation, alors qu'il faut attendre l'exécution du programme pour les premiers.

Dans un langage de classe typé, comme par exemple JAVA, le *polymorphisme* est contrôlé par l'héritage. Ainsi, une variable de type `Rectangle` désigne bien évidemment des occurrences d'objets de type `Rectangle`, mais pourra également désigner des objets de type `Carré`. Si la variable `r` est de type `Rectangle`, et la variable `c` de type `Carré`, l'affectation `r←c` est valide. Cette affectation se justifie puisqu'un carré est en fait un rectangle dont la largeur et la longueur sont égales. La relation d'héritage qui lie les rectangles et les carrés est vue comme une relation *est-un*. Un carré *est un* rectangle (spécialisé). En revanche, l'affectation inverse, `c←r`, n'est pas licite, puisqu'un rectangle n'est pas (nécessairement) un carré.

Le polymorphisme des langages de classe typés permet d'assouplir le système de type, tout en conservant un contrôle de type rigoureux. Imaginons que l'on veuille manipuler des formes géométriques diverses. Nous définirons la classe `Forme` pour représenter des formes géométriques quelconques. La classe `Rectangle` héritera de cette classe, puisqu'un rectangle est bien une forme géométrique. De même, nous définirons des classes pour représenter des ellipses et des cercles. L'arbre d'héritage de ces classes est donné par la figure 3.

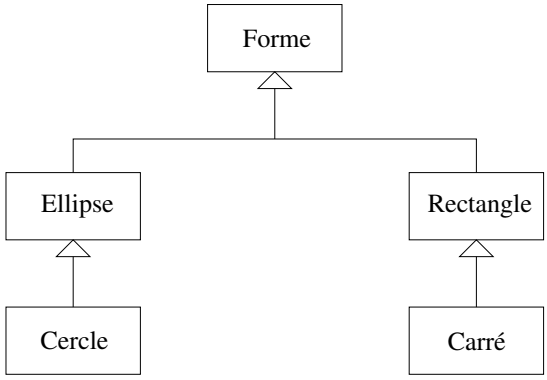

FIG. 3 Arbre d'héritage des figures géométriques.

Les éléments d'un tableau de type `Forme` pourront désigner, à tout moment, des ellipses, des cercles, des rectangles ou encore des carrés. Sans le polymorphisme, il aurait fallu déclarer autant de tableaux qu'il existe de formes géométriques.

```
t type tableau[ [1,max] ] de Forme

t[1] ← créer Rectangle(3,10)
t[2] ← créer Cercle(8)
...
t[1] ← t[2] {t[1] désigne (peut-être) un cercle}
```

Chacune des classes, qui hérite de `Forme`, est pourvue d'une méthode `afficher` qui produit un affichage spécifique de la forme qu'elle définit. S'il est clair qu'après la première affectation :

```
t[1] ← créer Rectangle(3,10)
```

l'instruction `t[1].afficher()` produit l'affichage d'un rectangle, il n'en va pas de même si cette même instruction est exécutée après la dernière affectation :

```
t[1] ← t[2]
```

Pour cette dernière, la méthode à appliquer ne peut être connue qu'à l'exécution du programme, au moment où l'on connaît la nature de l'objet qui a été affecté à `t[1]`, c'est-à-dire l'objet qu'il désigne réellement. Ce sera la méthode `afficher` de la classe `Cercle`, si `t[2]` désigne bien un cercle au moment de l'affectation. Lorsqu'il y a des redéfinitions de méthodes, c'est au moment de l'exécution que l'on connaît la méthode à appliquer. Elle est déterminée à partir de la forme dynamique de l'objet sur lequel elle s'applique. Ce mécanisme, appelé *liaison dynamique*, s'applique à des méthodes qui possèdent exactement les mêmes signatures, proposant dans des classes *différentes* d'une même ascendance, la mise en œuvre d'une même opération. Il a pour intérêt majeur une *utilisation* des méthodes redéfinies *indépendamment* des objets qui les définissent. On voit bien dans l'exemple précédent, qu'il est possible d'afficher ou de calculer le périmètre d'une forme sans se soucier de sa nature exacte.

5 CLASSES ABSTRAITES

Dans la section précédente, nous n'avons pas rédigé le corps des méthodes de la classe `Forme`. Puisque cette classe représente des formes quelconques, il est bien difficile d'écrire les méthodes `surface` ou `afficher`. Toutefois, ces méthodes doivent être nécessairement définies dans cette classe pour qu'il y ait polymorphisme ; la variable `t` est un tableau de `Forme` et `t[1].afficher()` doit être définie. Il est toujours possible de définir le corps des méthodes vide, mais alors rien ne garantit que les méthodes seront effectivement redéfinies par les classes héritières.

Une classe *abstraite* est une classe très générale qui décrit des propriétés qui ne seront définies que par des classes héritières, soit parce qu'elle ne sait pas comment le faire (*e.g.* la classe `Forme`), soit parce qu'elle désire proposer différentes mises en œuvres (). Nous définirons, par exemple, la classe `Forme` comme suit :

```
classe abstraite Forme
    public périmètre, surface, afficher
    {les méthodes abstraites}
    fonction périmètre() : réel
    fonction surface() : réel
    procédure afficher()
finclasse abstraite Forme
```

Les méthodes d'une telle classe sont appelées *méthodes abstraites*, et seuls les en-têtes sont spécifiés. Une classe abstraite ne peut être instanciée, il n'est donc pas possible de créer des objets de type `Forme`. De plus, les classes héritières (*e.g.* `Rectangle`) sont dans l'obligation de redéfinir les méthodes de la classe abstraite, sinon elles seront considérées elles-mêmes comme abstraites, et ne pourront donc pas être instanciées.

6 HÉRITAGE SIMPLE ET MULTIPLE

Il arrive fréquemment qu'une classe doive posséder les caractéristiques de plusieurs classes parentes distinctes. Une figure géométrique formée d'un carré avec en son centre un cercle d'un rayon égal à celui du côté du carré, pourrait être décrite par une classe qui hériterait à la fois des propriétés des carrés et des cercles (voir la figure 4). Cette classe serait par exemple contrainte par la relation *largeur = longueur = diamètre*.

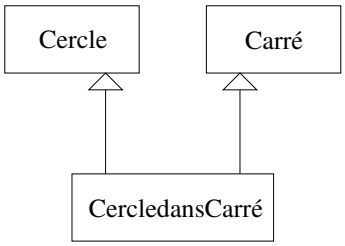

FIG. 4 Graphe d'héritage CercledansCarré.

Lorsqu'une classe ne possède qu'une seule classe parente, l'héritage est *simple*. En revanche, si une classe peut hériter de plusieurs classes parentes différentes, l'héritage est alors *multiple*. Avec l'héritage multiple, les relations d'héritage entre les classes ne définissent plus une simple arborescence, mais de façon plus générale un graphe, appelé *graphe d'héritage*[3].

L'héritage multiple introduit une complexité non négligeable dans le choix de la méthode à appliquer en cas de conflit de noms ou d'héritage répété. Pour le programmeur, le choix d'une méthode à appliquer peut ne pas être évident. C'est pour cela que certains langages de programmation, comme JAVA[4], ne le permettent pas.

7 HÉRITAGE ET ASSERTIONS

Le mécanisme d'héritage introduit de nouvelles règles pour la définition des assertions des classes héritières et des méthodes qu'elles comportent.

7.1 Assertions sur les classes héritières

L'invariant d'une classe héritière est la conjonction des invariants de ses classes parentes et de son propre invariant. Dans notre exemple, l'invariant de la classe `Carré` est celui de la classe `Rectangle`, *i.e.* la largeur et la longueur d'un rectangle doivent être positives ou nulles, *et* de son propre invariant, *i.e.* ces deux longueurs doivent être égales.

7.2 Assertions sur les méthodes

Les règles de définition des antécédents et des conséquents sur les méthodes doivent être complétées dans le cas particulier de la redéfinition. Nous prendrons ici les règles données par B. MEYER [Mey97].

Une assertion A est plus forte qu'une assertion B, si A implique B. Inversement, nous dirons que B est plus faible. Lors d'une redéfinition d'une méthode m, que nous appellerons m', il faudra que :

(1) l'antécédent de m' soit plus faible ou égal que celui de m ;

(2) le conséquent de m' soit plus fort ou égal que celui de m.

Pour comprendre cette règle, il faut la voir à la lumière de la liaison dynamique. Une variable déclarée de type classe A peut appeler la méthode m, mais exécuter sa redéfinition m' dans la classe héritière B sous l'effet de la liaison dynamique. Cela indique que toute assertion qui s'applique à m *doit* également s'appliquer à m'. Aussi, la règle (1) indique que m' doit accepter l'antécédent de m, et la règle (2) que m' doit également vérifier le conséquent de m.

8 RELATION D'HÉRITAGE OU DE CLIENTÈLE

Lors de la construction d'un programme, comment choisir les relations à établir entre les classes ? Une classe A doit-elle hériter d'une classe B, ou en être la cliente ? Une première réponse est de dire que si on peut appliquer la relation *est-un*, sans doute faudra-t-il utiliser l'héritage. Dans notre exemple, un carré *est-un* rectangle particulier dont la largeur et la longueur sont égales. La classe `Carré` hérite de la classe `Rectangle`. En revanche, si c'est une relation *a-un* qui doit s'appliquer, il faudra alors établir une relation de clientèle. Une voiture *a-un* volant, une école *a-des* élèves. La classe `Voiture`, qui représente des automobiles, possédera un attribut `volant` qui le décrit. De même, les élèves d'une école peuvent être représentés par la classe `Élèves`, et la classe `École` possédera un attribut pour désigner tous les élèves de l'école.

Cette règle possède l'avantage d'être simple et nous l'utiliserons chaque fois que cela est possible. Toutefois, elle ne pourra être appliquée systématiquement, et nous verrons par la suite des cas où elle devra être mise en défaut.

9 L'HÉRITAGE EN JAVA

En JAVA, l'héritage est simple et toutes les classes possèdent *implicitement* un ancêtre commun, la classe `Object`. On retrouve dans ce langage les concepts exposés précédemment, et dans cette section, nous n'évoquerons que ses spécificités.

Les classes héritières, ou *sous-classes*, comportent dans leur en-tête le mot-clé **`extends`** suivi du nom de la classe parente. Le constructeur d'une sous-classe peut faire appel à un

constructeur de sa classe parente, la *super-classe* appelée **super**. S'il ne le fait pas, un appel implicite au constructeur par défaut de la classe parente, c'est-à-dire **super**(), aura systématiquement lieu. Notez que le constructeur par défaut de la classe mère doit alors exister. La classe Carré s'écrit en JAVA :

```
public class Carré extends Rectangle {
    /** Invariant de classe : longueur = largeur ⩾ 0 */
    // le constructeur
    public Carré(double c) {
         // appel du constructeur de Rectangle
         super(c,c);
    }
    public void changerCôté(double côté)
    // Rôle : met à jour le côté du carré courant
    {
        changerLargeur(côté);
        changerLongueur(côté);
    }
    public String toString()
    // Rôle : convertit le carré courant en chaîne de caractères
    {
        return "carré de côté égal à" + largeur;
    {
} // fin classe Carré
```

La création d'un carré c de côté 7 et l'affichage de sa surface s'écriront comme suit :

```
Carré c = new Carré(7);
System.out.println(c.surface());
```

La redéfinition des méthodes dans les sous-classes ne peut se faire qu'avec des méthodes qui possèdent exactement les mêmes signatures. La liaison dynamique est donc mise en œuvre sur des méthodes qui possèdent les mêmes en-têtes et qui diffèrent par leurs instructions, comme par exemple la méthode toString des classes Rectangle et Carré.

Les classes abstraites sont introduites par le mot-clé **abstract**, de même que les méthodes. Notez que seule une partie des méthodes peut être déclarée abstraite, la classe demeurant toutefois abstraite. La classe Forme possède la déclaration suivante :

```
abstract class Forme {
    public abstract double périmètre();
    public abstract double surface();
}
```

Remarquez l'absence de la méthode toString, puisque celle-ci est héritée de la classe Object.

Contrairement aux classes abstraites, les *interfaces* sont des classes dont *toutes* les méthodes sont *implicitement* abstraites et qui ne peuvent posséder d'attributs, à l'exception de constantes. Les interfaces permettent, d'une part, une forme simplifiée de l'héritage multiple, et d'autre part la *généricité*. Nous reparlerons de ces deux notions plus loin, à partir du chapitre 16.

L'interface de programmation d'application de JAVA (API[6]) est une hiérarchie de classes qui offrent aux programmeurs des classes préfabriquées pour manipuler les fichiers, construire des interfaces graphiques, établir des communications réseaux, *etc*.

La classe `Object` est au sommet de cette hiérarchie. Elle possède en particulier deux méthodes `clone()` et `equals(Object o)`. La première permet de dupliquer l'objet courant, et la seconde de comparer si l'objet passé en paramètre est égal à l'objet courant. Ces méthodes sont nécessaires puisque les opérations d'affectation et d'égalité mettant en jeu deux opérandes de type classe manipulent des références et non pas les objets eux-mêmes.

Il n'est pas question de présenter ces classes ici ; ce n'est d'ailleurs pas l'objet de cet ouvrage. Toutefois, il faut mentionner que les types simples primitifs du langage possèdent leur équivalent objet dont la correspondance est donnée par la table 1.

Type primitif	*Classe correspondante*
byte	`Byte`
short	`Short`
int	`Integer`
long	`Long`
float	`Float`
double	`Double`
boolean	`Boolean`
char	`Character`

TAB. 1 Correspondance types primitifs – classes.

Notez que depuis sa version 5.0, le langage permet des conversions implicites entre un type primitif et son équivalent objet, et réciproquement. JAVA appelle ce mécanisme *AutoBoxing*. Par exemple, il est désormais possible d'écrire :

```
Character c = 'a';
int i = new Integer(10);
```

➤ Règles de visibilité

Nous avons déjà vu qu'un membre d'une classe pouvait être qualifié **`public`**, pour le rendre visible par n'importe quelle classe, et **`private`**, pour restreindre sa visibilité à sa classe de définition. Le langage JAVA propose une troisième qualification, **`protected`**, qui rend visible le membre par toutes les classes héritières de sa classe de définition.

En fait, les membres qualifiés **`protected`** sont visibles par les héritiers de la classe de définition, mais également par toutes les classes du même paquetage (**`package`**). En JAVA, un *paquetage* est une collection de classes placée dans des fichiers regroupés dans un même répertoire ou dossier, selon la terminologie du système d'exploitation utilisé. Un paquetage regroupe des classes qui possèdent des caractéristiques communes, comme par exemple le paquetage `java.io` pour toutes les entrées/sorties. La directive **`import`** suivie du nom d'un paquetage permet de dénoter les noms que définit ce dernier, sans les préfixer par le nom du paquetage. Par exemple, les deux déclarations de variables suivantes sont équivalentes :

```
import java.util.Random;      ou alors      java.util.Random x;
Random x;
```

Depuis la version 5.0, JAVA spécialise la directive **`import`** pour l'accès non qualifié aux objets déclarés **`static`** d'une classe. Ainsi, au lieu d'écrire :

```
i1 = Math.sqrt(-Δ)/(2*a);
```

on pourra écrire :

```
import static java.lang.Math.*;
...
i1 = sqrt(-Δ)/(2*a);
```

Des règles de visibilité sont également définies pour les classes. Une déclaration d'une classe préfixée par le mot-clé **`public`** rendra la classe visible par n'importe quelle classe depuis n'importe quel paquetage. Si ce mot-clé n'apparaît pas, la visibilité de la classe est alors limitée au paquetage. Les classes dont on veut limiter la visibilité au paquetage sont, en général, des classes auxiliaires nécessaires au bon fonctionnement des classes publiques du paquetage, mais inutiles en dehors.