Déclaration et création de tableaux

En programmation, on parle de tableau pour désigner un ensemble d'éléments de même type désignés par un nom unique, chaque élément étant repéré par un indice précisant sa position au sein de l'ensemble.

Comme tous les langages, Java permet de manipuler des tableaux mais nous verrons qu'il fait preuve d'originalité sur ce point. En particulier, les tableaux sont considérés comme des objets et les tableaux à plusieurs indices s'obtiennent par composition de tableaux.

Nous commencerons par voir comment déclarer puis créer des tableaux, éventuellement les initialiser. Nous étudierons ensuite la manière de les utiliser, soit au niveau de chaque élément, soit à un niveau global. Nous examinerons alors la transmission de tableaux en argument ou en valeur de retour d'une méthode. Enfin, nous aborderons la création et l'utilisation des tableaux à plusieurs indices.

1.1 Introduction

Considérons cette déclaration :

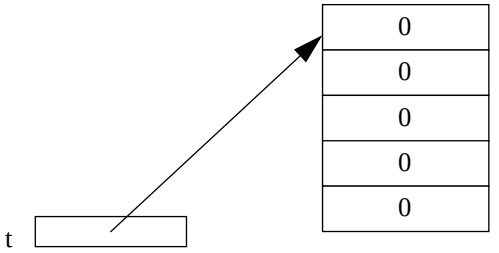
```
int t[] ;
```

Elle précise que *t* est destiné à contenir la référence à un tableau d'entiers. Vous constatez qu'aucune dimension ne figure dans cette déclaration et, pour l'instant, aucune valeur n'a été attribuée à *t*. Cette déclaration est en fait très proche de celle de la référence à un objet[1].

On crée un tableau comme on crée un objet, c'est-à-dire en utilisant l'opérateur *new*. On précise à la fois le type des éléments, ainsi que leur nombre (dimension du tableau), comme dans :

```
t = new int[5] ;     // t fait référence à un tableau de 5 entiers
```

Cette instruction alloue l'emplacement nécessaire à un tableau de 5 éléments de type *int* et en place la référence dans *t*. Les 5 éléments sont initialisés par défaut (comme tous les champs d'un objet) à une valeur "nulle" (0 pour un *int*). On peut illustrer la situation par ce schéma :



1.2 Déclaration de tableaux

La déclaration d'une référence à un tableau précise donc simplement le type des éléments du tableau. Elle peut prendre deux formes différentes ; par exemple, la déclaration précédente :

```
int t[] ;
```

peut aussi s'écrire :

```
int [] t ;
```

En fait, la différence entre les deux formes devient perceptible lorsque l'on déclare plusieurs identificateurs dans une même instruction. Ainsi,

```
int [] t1, t2 ;    // t1 et t2 sont des références à des tableaux d'entiers
```

est équivalent à :

```
int t1[], t2[] ;
```

La première forme permet le mélange de tableaux de type *T* et de variables de type *T* :

```
int t1[], n, t2[] ;  // t1 et t2 sont des tableaux d'entiers, n est entier
```

En Java, les éléments d'un tableau peuvent être d'un type primitif ou d'un type objet. Par exemple, si nous avons défini le type classe *Point*, ces déclarations sont correctes :

```
Point tp [] ;      // tp est une référence à un tableau d'objets de type Point
Point a, tp[], b ; // a et b sont des références à des objets de type Point
                   // tp est une référence à un tableau d'objets de type Point
```

1. En particulier, elle sera soumise aux mêmes règles d'initialisation : valeur *null* s'il s'agit d'un champ d'objet, initialisation obligatoire avant toute utilisation dans les autres cas.

Remarque

Une déclaration de tableau ne doit pas préciser de dimensions. Cette instruction sera rejetée à la compilation :

```
int t[5] ;    // erreur : on ne peut pas indiquer de dimension ici
```

1.3 Création d'un tableau

Nous avons vu comment allouer l'emplacement d'un tableau comme celui d'un objet à l'aide de l'opérateur *new*. On peut aussi utiliser un *initialiseur* au moment de la déclaration du tableau, comme on le fait pour une variable d'un type primitif.

1.3.1 Création par l'opérateur new

La valeur de l'expression fournie à l'opérateur *new* n'est calculée qu'au moment de l'exécution du programme. Elle peut donc différer d'une fois à l'autre, contrairement à ce qui se produit dans le cas des langages fixant la dimension lors de la compilation. Voici un exemple :

```
System.out.print ("taille voulue ? ") ;
int n = Clavier.lireInt() ;
int t[] = new int [n] ;
```

Notez cependant que l'objet tableau une fois créé ne pourra pas voir sa taille modifiée. En revanche, comme n'importe quelle référence à un objet, la référence contenue dans *t* pourra très bien évoluer au fil de l'exécution et désigner finalement des tableaux différents, éventuellement de tailles différentes.

Enfin, sachez qu'il est permis de créer un tableau de taille nulle (qu'on ne confondra pas avec une référence nulle). En revanche, l'appel de *new* avec une valeur négative conduira à une exception *NegativeArraySizeException*.

1.3.2 Utilisation d'un initialiseur

Lors de la déclaration d'une référence de tableau, on peut fournir une liste d'expressions entre accolades, comme dans :

```
int n, p ;
   ...
int t[] = {1, n, n+p, 2*p, 12} ;
```

Cette instruction crée un tableau de 5 entiers ayant les valeurs des expressions mentionnées et en place la référence dans *t*. Elle remplace les instructions suivantes :

```
int n, p, t[] ;
   .....
t = new int[5] ;
t[0] = 1 ; t[1] = n ; t[2] = n+p ; t[3] = 2*p ; t[4] = 12 ;
```

Java se sert du nombre d'expressions figurant dans l'initialiseur pour en déduire la taille du tableau à créer. Notez que ces expressions n'ont pas besoin d'être des expressions constantes ; il suffit simplement qu'elles soient calculables au moment où l'on exécute l'opérateur *new*.

Remarque

La notation {.....} n'est utilisable que dans une déclaration. L'instruction suivante serait incorrecte :

```
t = {1, n, n+p, 2*p, 12} ;
```

2 Utilisation d'un tableau

En Java, on peut utiliser un tableau de deux façons différentes :

- en accédant individuellement à chacun de ses éléments,
- en accédant globalement à l'ensemble du tableau.

2.1 Accès individuel aux éléments d'un tableau

On peut manipuler un élément de tableau comme on le ferait avec n'importe quelle variable ou n'importe quel objet du type de ses éléments. On désigne un élément particulier en plaçant entre crochets, à la suite du nom du tableau, une expression entière nommée indice indiquant sa position. Le premier élément correspond à l'indice 0 (et non 1).

```
int t[] = new int[5] ;
   .....
t[0] = 15 ;  // place la valeur 15 dans le premier élément du tableau t
   .....
t[2]++ ;     // incrémente de 1 le troisième élément de t
   .....
System.out.println (t[4]) ;  // affiche la valeur du dernier élément de t
```

Si, lors de l'exécution, la valeur d'un indice est négative ou trop grande par rapport à la taille du tableau, on obtient une erreur d'exécution. Plus précisément, il y a déclenchement d'une *exception* de type *ArrayIndexOutOfBoundsException*. Nous apprendrons au Chapitre 10 qu'il est possible d'intercepter une telle exception. Si nous le faisons pas, nous aboutissons simplement à l'arrêt de l'exécution du programme ; en fenêtre console s'affichera un message d'erreur.

Voici un exemple complet de programme utilisant un tableau de flottants pour déterminer le nombre d'élèves d'une classe ayant une note supérieure à la moyenne de la classe[1].

1. Notez bien que s'il s'agissait seulement de déterminer la moyenne de la classe, il ne serait pas indispensable d'utiliser un tableau.

```
public class Moyenne
{ public static void main (String args[])
  {  int i, nbEl, nbElSupMoy ;
     double somme ;
     double moyenne ;
     System.out.print ("Combien d'eleves ") ;
     nbEl = Clavier.lireInt();
     double notes[] = new double[nbEl] ;
     for (i=0 ; i<nbEl ; i++)
      { System.out.print ("donnez la note numero " + (i+1) + " : " ) ;
        notes[i] = Clavier.lireDouble() ;
      }
     for (i=0, somme=0 ; i<nbEl ; i++)  somme += notes[i] ;
     moyenne = somme / nbEl ;
     System.out.println ("\nmoyenne de la classe  " + moyenne) ;
     for (i=0, nbElSupMoy=0 ; i<nbEl ; i++ )
       if (notes[i] > moyenne) nbElSupMoy++ ;
     System.out.println (nbElSupMoy + " eleves ont plus de cette moyenne") ;
  }
}
```

```
Combien d'eleves 5
donnez la note numero 1 : 12
donnez la note numero 2 : 14.5
donnez la note numero 3 : 10
donnez la note numero 4 : 9
donnez la note numero 5 : 16

moyenne de la classe  12.3
2 eleves ont plus de cette moyenne
```

Exemple d'utilisation d'un tableau

Vous constatez que la possibilité de définir la dimension du tableau au moment de sa création permet de travailler avec un nombre d'élèves quelconques.

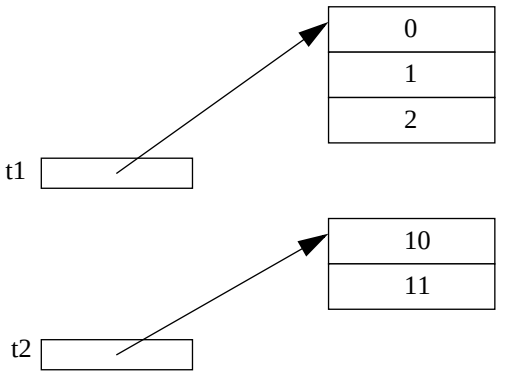
2.2 Affectation de tableaux

Le paragraphe précédent vous a montré comment accéder individuellement à chacun des éléments d'un tableau existant. Java permet aussi de manipuler globalement des tableaux, par le biais d'affectations de leurs références.

Considérons ces instructions qui créent deux tableaux d'entiers en plaçant leurs références dans *t1* et *t2* :

```
int [] t1 = new int[3] ;
for (int i=0 ; i<3 ; i++) t1[i] = i ;
int [] t2 = new int[2] ;
for (int i=0 ; i<2 ; i++) t2[i] = 10 + i ;
```

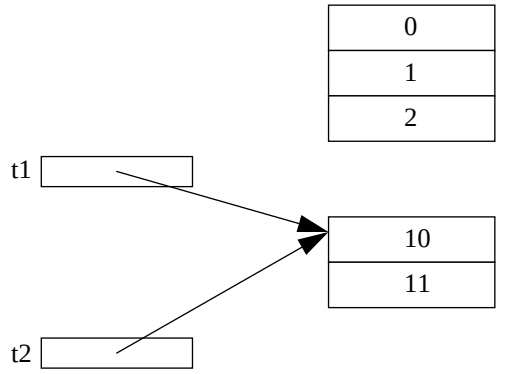
La situation peut être schématisée ainsi :



Exécutons maintenant l'affectation :

```
t1 = t2 ;     // la référence contenue dans t2 est recopiée dans t1
```

Nous aboutissons à cette situation :



Dorénavant, *t1* et *t2* désignent le même tableau. Ainsi, avec :

```
t1[1] = 5 ;
System.out.println (t2[1]) ;
```

on obtiendra l'affichage de la valeur 5, et non 10.

Si l'objet que constitue le tableau de trois entiers anciennement désigné par *t1* n'est plus référencé par ailleurs, il deviendra candidat au ramasse-miettes.

Il est très important de noter que l'affectation de références de tableaux n'entraîne aucune recopie des valeurs des éléments du tableau. On retrouve exactement le même phénomène que pour l'affectation d'objets.

Remarques

1 Un objet tableau tel que celui crée par *new int[3]* a une dimension fixée pour toute la durée du programme (même si celle-ci est fixée lors de l'exécution). En revanche, l'objet référencé par *t1* pouvant évoluer au fil de l'exécution, sa dimension peut elle aussi évoluer. Cependant, il ne s'agit pas de tableaux dynamiques au sens usuel du terme. De tels tableaux peuvent être obtenus en Java en recourant à la classe *Vector* du paquetage *java.util*.

2 Si les éléments de deux tableaux ont des types compatibles par affectation, les références correspondantes ne sont pas pour autant compatibles par affectation. Si l'on considère par exemple :

```
int te[] tEnt ;
float tf [] tFlot ;
```

il n'est pas possible d'écrire :

```
tFlot = tEnt ;
```

et ce, bien qu'un *int* puisse être affecté à un *float*.

En revanche, une telle compatibilité existera entre un tableau d'objets d'une classe et un tableau d'objets de sa classe de base, comme nous le verrons au chapitre consacré à l'héritage.

2.3 La taille d'un tableau : length

La déclaration d'une référence de tableau n'en précise pas la taille et nous avons vu que cette dernière peut évoluer au fil de l'exécution d'un programme. Le champ *length* permet de connaître le nombre d'éléments d'un tableau de référence donnée :

```
int t[] = new int[5] ;
System.out.println ("taille de t : " + t.length) ;   // affiche 5
t = new int[3] ;
System.out.println ("taille de t : " + t.length) ;   // affiche 3
```

Remarque

Notez bien qu'on écrit *t.length* et non *t.length()* car *length* s'utilise comme s'il s'agissait d'un champ public de l'objet tableau *t* et non d'une méthode.

2.4 Exemple de tableau d'objets

Comme nous l'avons déjà dit, les éléments d'un tableau peuvent être de type quelconque, et pas seulement d'un type primitif comme dans nos précédents exemples. Voici un exemple de programme utilisant un tableau d'objets de type *Point* :

```
public class TabPoint
{ public static void main (String args[])
  { Point [] tp ;
    tp = new Point[3] ;
    tp[0] = new Point (1, 2) ;
    tp[1] = new Point (4, 5) ;
    tp[2] = new Point (8, 9) ;
    for (int i=0 ; i<tp.length ; i++)
      tp[i].affiche() ;
  }
}
class Point
{ public Point(int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void affiche ()
  { System.out.println ("Point : " + x + ", " + y) ;
  }
  private int x, y ;
}
```

```
Point : 1, 2
Point : 4, 5
Point : 8, 9
```

Exemple d'utilisation d'un tableau d'objets (de type Point*)*

3 Tableau en argument ou en retour

Lorsqu'on transmet un nom de tableau en argument d'une méthode, on transmet en fait (une copie de) la référence au tableau. La méthode agit alors directement sur le tableau concerné et non sur une copie. On retrouve exactement le même phénomène que pour les objets (paragraphe 9.3 du chapitre 6).

Voici un exemple de deux méthodes statiques (définies dans une class utilitaire nommée *Util*) permettant :

- d'afficher les valeurs des éléments d'un tableau d'entiers (méthode *affiche*),
- de mettre à zéro les éléments d'un tableau d'entier (méthode *raz*).

```
public class TabArg
{ public static void main (String args[])
  { int t[] = { 1, 3, 5, 7} ;
    System.out.print ("t avant : ") ;
    Util.affiche (t) ;
    Util.raz (t) ;
```

```
    System.out.print ("\nt apres : ") ;
    Util.affiche (t) ;
  }
}
class Util
{ static void raz (int t[])
  { for (int i=0 ; i<t.length ; i++)
      t[i] = 0 ;
  }
  static void affiche (int t[])
  { for (int i=0 ; i<t.length ; i++)
      System.out.print (t[i] + " ") ;
  }
}
```

```
t avant : 1 3 5 7
t apres : 0 0 0 0
```

Exemple de fonctions recevant un tableau en argument

Les mêmes reflexions s'appliquent à un tableau fourni en valeur de retour. Par exemple, la méthode suivante fournirait en résultat un tableau formé des *n* premiers nombres entiers :

```
public static int[] suite (int n)
{ int[] res = new int[n] ;
  for (int i=0 ; i<n ; i++) res[i] = i+1 ;
  return res ;
}
```

Un appel de *suite* fournira une référence à un tableau dont on pourra éventuellement modifier les valeurs des éléments.

En C++

En C++, un tableau n'est pas un objet. Il n'existe pas d'équivalent du mot clé *length*. La transmission d'un tableau en argument correspond à son adresse ; elle est généralement accompagnée d'un second argument en précisant la taille (ou le nombre d'éléments qu'on souhaite traiter à partir d'une adresse donnée, laquelle peut éventuellement correspondre à un élément différent du premier).

4 Les tableaux à plusieurs indices

De nombreux langages disposent de la notion de tableau à plusieurs indices. Par exemple, un tableau à deux indices permet de représenter une matrice mathématique.

Java ne dispose pas d'une telle notion. Néanmoins, il permet de la "simuler" en créant des tableaux de tableaux, c'est-à-dire des tableaux dont les éléments sont eux-mêmes des

tableaux. Comme nous allons le voir, cette possibilité s'avère en fait plus riche que celle de tableaux à plusieurs indices offerte par les autres langages. Elle permet notamment de disposer de tableaux irréguliers, c'est-à-dire dans lesquels les différentes lignes[1] pourront être de taille différente. Bien entendu, on pourra toujours se contenter du cas particulier dans lequel toutes les lignes auront la même taille et, donc, manipuler l'équivalent des tableaux à deux indices classiques.

4.1 Présentation générale

Premier exemple

Ces trois déclarations sont équivalentes :

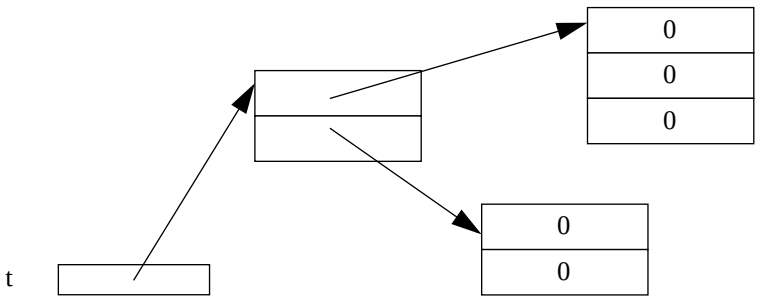
```
int t [] [] ;
int [] t [] ;
int [] [] t ;
```

Elles déclarent que *t* est une référence à un tableau, dans lequel chaque élément est lui-même une référence à un tableau d'entiers. Pour l'instant, comme à l'accoutumée, aucun tableau de cette sorte n'existe encore.

Mais considérons la déclaration :

```
int t [] [] = {  new int [3], new int [2] } ;
```

L'initialiseur de *t* comporte deux éléments dont l'évaluation crée un tableau de 3 entiers et un tableau de 2 entiers. On aboutit à cette situation (les éléments des tableaux d'entiers sont, comme d'habitude, initialisés à 0) :



Dans ces conditions, on voit que :

- la notation *t[0]* désigne la référence au premier tableau de 3 entiers,
- la notation *t[0][1]* désigne le deuxième élément de ce tableau (les indices commencent à 0),
- la notation *t[1]* désigne la référence au second tableau de 2 entiers,
- la notation *t[1][i-1]* désigne le i^{ème} élément de ce tableau,

1. Malgré son ambiguïté, nous utilisons le terme ligne dans son acceptation habituelle, c'est-à-dire pour désigner en fait, dans un tableau à deux indices, l'ensemble des éléments ayant la même valeur du premier indice.

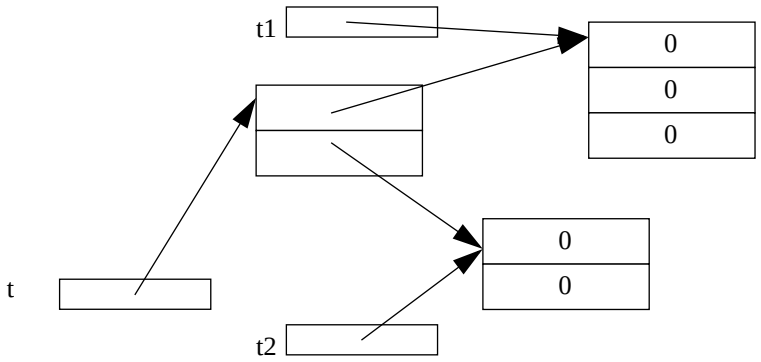
- l'expression *t.length* vaut 2,
- l'expression *t[0].length* vaut 3,
- l'expression *t[1].length* vaut 2.

Second exemple

On peut aboutir à une situation très proche de la précédente en procédant ainsi :

```
int t[][] ;
t = new int [2] [] ;        // création d'un tableau de 2 tableaux d'entiers
int [] t1 = new int [3] ;   // t1 = référence à un tableau de 3 entiers
int [] t2 = new int [2] ;   // t2 = référence à un tableau de 2 entiers
t[0] = t1 ; t[1] = t2 ;     // on range ces deux références dans t
```

Hormis les différences concernant les instructions, on voit qu'on a créé ici deux variables supplémentaires *t1* et *t2* contenant les références aux deux tableaux d'entiers. La situation peut être illustrée ainsi :



En C++

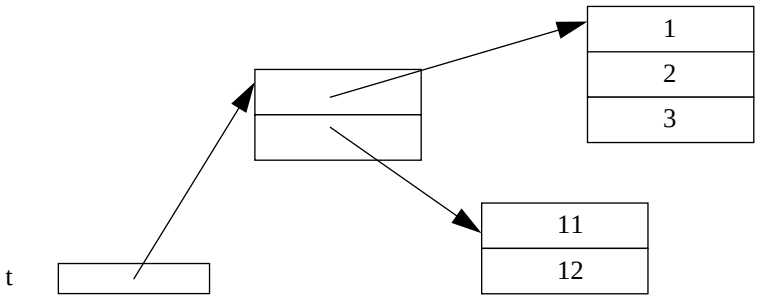
En C++, les éléments d'un tableau à deux indices sont contigus en mémoire. On peut, dans certains cas, considérer un tel tableau comme un (grand) tableau à un indice. Ce n'est pas le cas en Java.

4.2 Initialisation

Dans le premier exemple, nous avons utilisé un initialiseur pour les deux références à introduire dans le tableau *t* ; autrement dit, nous avons procédé comme pour un tableau à un indice. Mais comme on s'y attend, les initialiseurs peuvent tout à fait s'imbriquer, comme dans cet exemple :

```
int t[] [] = { {1, 2, 3}, {11, 12} } ;
```

Il correspond à ce schéma :



4.3 Exemple

Voici un exemple de programme complet utilisant deux méthodes statiques définies dans une classe *Util*, permettant :

- d'afficher les valeurs des éléments d'un tableau d'entiers à deux indices, ligne par ligne (méthode *affiche*),
- de mettre à zéro les éléments d'un tableau d'entiers à deux indices (méthode *raz*).

```
class Util
{ static void raz (int t[] [])
  { int i, j ;
    for (i= 0 ; i<t.length ; i++)
      for (j=0 ; j<t[i].length ; j++)
         t[i] [j] = 0 ;
  }
  static void affiche (int t[] [])
  { int i, j ;
    for (i= 0 ; i<t.length ; i++)
      { System.out.print ("ligne de rang " + i + "= ") ;
        for (j=0 ; j<t[i].length ; j++)
           System.out.print (t[i] [j] + " ") ;
        System.out.println() ;
      }
  }
}

public class Tab2ind1
{ public static void main (String args[])
  { int t[] [] = { {1, 2, 3}, {11, 12}, {21, 22, 23, 24} } ;
    System.out.println ("t avant raz : ") ;
    Util.affiche(t) ;
    Util.raz(t) ;
```

```
    System.out.println ("t apres raz : ") ;
    Util.affiche(t) ;
  }
}
```

```
t avant raz :
ligne de rang 0= 1 2 3
ligne de rang 1= 11 12
ligne de rang 2= 21 22 23 24
t apres raz :
ligne de rang 0= 0 0 0
ligne de rang 1= 0 0
ligne de rang 2= 0 0 0 0
```

Exemple de méthodes (statiques) recevant en argument un tableau à deux indices

4.4 Cas particulier des tableaux réguliers

Qui peut le plus peut le moins. Autrement dit, rien n'empêche que, dans un tableau à deux indices, toutes les lignes aient la même taille. Par exemple, si l'on souhaite disposer d'une matrice de *NLIG* lignes et de *NCOL* colonnes, on pourra toujours procéder ainsi :

```
int t[] [] = new int [NLIG] [] ;
int i ;
for (i=0 ; i<NLIG ; i++) t[i] = new int[NCOL] ;
```

Mais Java permet alors d'écrire les choses plus simplement :

```
int t[] [] = new int [NLIG] [NCOL] ;
```

Bien entendu, il est possible d'utiliser un tel tableau sans recourir au mot clé *length*, comme dans les autres langages. Par exemple, voici comment nous pourrions mettre à zéro tous les éléments de *t* :

```
for (int i =0 ; i<NLIG ; i++)
  for (int j=0 ; j<NCOL ; j++)
    t[i][j] = 0 ;
```

Toutefois, il ne faudra pas perdre de vue que, même dans ce cas, les valeurs de *t* comme celles de *t[i]* sont des références dont la valeur peut être modifiée au cours de l'exécution. En particulier, lorsqu'on écrit une méthode destinée à un tel tableau régulier, on peut être tenté d'y utiliser classiquement un nombre de colonnes défini par exemple comme étant le nombre d'éléments de la première ligne du tableau. Mais, dans ce cas, on court le risque d'appeler par erreur cette méthode sur un tableau irrégulier, avec toutes les conséquences désastreuses qu'on peut imaginer. Les mêmes considérations valent si l'on cherche à transmettre *NLIG* et *NCOL* en arguments.

8

L'héritage

Comme nous l'avons déjà signalé au Chapitre 1, le concept d'héritage constitue l'un des fondements de la Programmation Orientée Objets. Il est notamment à l'origine des possibilités de réutilisation des composants logiciels que sont les classes. En effet, il permet de définir une nouvelle classe, dite *classe dérivée*, à partir d'une classe existante dite *classe de base*. Cette nouvelle classe hérite d'emblée des fonctionnalités de la classe de base (champs et méthodes) qu'elle pourra modifier ou compléter à volonté, sans qu'il soit nécessaire de remettre en question la classe de base.

Cette technique permet donc de développer de nouveaux outils en se fondant sur un certain acquis, ce qui justifie le terme d'héritage. Comme on peut s'y attendre, il sera possible de développer à partir d'une classe de base, autant de classes dérivées qu'on le désire. De même, une classe dérivée pourra à son tour servir de classe de base pour une nouvelle classe dérivée.

Comme on le verra au Chapitre 11, l'héritage constitue en outre l'un des pilliers de la programmation événementielle. En effet, la moindre application nécessitera de créer des classes dérivées des classes de la bibliothèque standard, en particulier de celles appartenant aux paquetages standard *java.awt*, *java.awt.event*, *javax.swing*.

Nous commencerons par vous présenter la notion d'héritage et sa mise en œuvre en Java. Nous verrons alors ce que deviennent les droits d'accès aux champs et méthodes d'une classe dérivée. Puis nous ferons le point sur la construction et l'initialisation des objets dérivés. Nous montrerons ensuite comment une classe dérivée peut redéfinir une méthode d'une classe de base et nous préciserons les interférences existant entre cette notion et celle de surdéfinition.

Puis nous présenterons la notion la plus fondamentale de Java, le polymorphisme, et nous exposerons ses règles. Après avoir montré que toute classe dérive d'une "super-classe" nommée *Object*, nous définirons ce qu'est une classe abstraite et l'intérêt qu'elle peut présenter.

Nous traiterons enfin des interfaces dont nous verrons qu'elles remplacent avantageusement l'héritage multiple de certains langages et qu'elle facilitent la tâche de réalisation des classes en imposant le respect d'un certain contrat.

1 La notion d'héritage

Nous allons voir comment mettre en œuvre l'héritage en Java, à partir d'un exemple simple de classe ne comportant pas encore de constructeur. Supposez que nous disposions de la classe *Point* suivante (pour l'instant, peu importe qu'elle ait été déclarée publique ou non) :

```
class Point
{
  public void initialise (int abs, int ord)
  { x = abs ; y = ord ;
  }

  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }

  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}
```

Une classe de base Point

Imaginons que nous ayons besoin d'une classe *Pointcol*, destinée à manipuler des points colorés d'un plan. Une telle classe peut manifestement disposer des mêmes fonctionnalités que la classe *Point*, auxquelles on pourrait adjoindre, par exemple, une méthode nommée *colore*, chargée de définir la couleur. Dans ces conditions, nous pouvons chercher à définir la classe *Pointcol* comme dérivée de la classe *Point*. Si nous prévoyons, outre la méthode *colore*, un membre nommé *couleur*, de type *byte*, destiné à représenter la couleur d'un point, voici comment pourrait se présenter la définition de la classe *Pointcol* (ici encore, peu importe qu'elle soit publique ou non) :

```
class Pointcol extends Point    // Pointcol dérive de Point
{ public void colore (byte couleur)
  { this.couleur = couleur ;
  }
  private byte couleur ;
}
```

Une classe Poincol, *dérivée de* Point

La mention *extends Point* précise au compilateur que la classe *Pointcol* est une classe dérivée de *Point*.

Disposant de cette classe, nous pouvons déclarer des variables de type *Pointcol* et créer des objets de ce type de manière usuelle, par exemple :

```
Pointcol pc ;   // pc contiendra une référence à un objet de type Pointcol
Pointcol pc2 = new Pointcol() ;  // pc2 contient la référence à un objet de
        // type Pointcol créé en utilisant le pseudo-constructeur par défaut
pc = new Pointcol() ;
```

Un objet de type *Pointcol* peut alors faire appel :

- aux méthodes publiques de *Pointcol*, ici *colore* ;
- mais aussi aux méthodes publiques de *Point* : *initialise*, *deplace* et *affiche*.

D'une manière générale, **un objet d'une classe dérivée accède aux membres publics de sa classe de base**, exactement comme s'ils étaient définis dans la classe dérivée elle-même.

Voici un petit programme complet illustrant ces possibilités (pour l'instant, la classe *Pointcol* est très rudimentaire ; nous verrons plus loin comment la doter d'autres fonctionnalités indispensables). Ici, nous créons à la fois un objet de type *Pointcol* et un objet de type *Point*.

```
   // classe de base
class Point
{ public void initialise (int abs, int ord)
  { x = abs ; y = ord ;
  }
  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }
  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}
   // classe derivee de Point
class Pointcol extends Point
{ public void colore (byte couleur)
  { this.couleur = couleur ;
  }
  private byte couleur ;
}
   // classe utilisant Pointcol
public class TstPcol1
{ public static void main (String args[])
  { Pointcol pc = new Pointcol() ;
    pc.affiche() ;
    pc.initialise (3, 5) ;
    pc.colore ((byte)3) ;
    pc.affiche() ;
```

```
    pc.deplace (2, -1) ;
    pc.affiche() ;
    Point p = new Point() ; p.initialise (6, 9) ;
    p.affiche() ;
  }
}
```

```
Je suis en 0 0
Je suis en 3 5
Je suis en 5 4
Je suis en 6 9
```

Exemple de création et d'utilisation d'une classe Pointcol *dérivée de* Point

Remarques

1 Une classe de base peut aussi se nommer une super-classe ou tout simplement une classe. De même, une classe dérivée peut aussi se nommer une sous-classe. Enfin, on peut parler d'héritage ou de dérivation, ou encore de sous-classement.

2 Ici, nous avons regroupé au sein d'un unique fichier source la classe *Point*, la classe *Pointcol* et une classe contenant la méthode *main*. Nous n'avons donc pas pu déclarer publiques les classes *Point* et *Pointcol*. En pratique, il en ira rarement ainsi : au minimum, la classe *Point* appartiendra à un fichier différent. Mais cela ne change rien aux principes de l'héritage ; seuls n'interviendront que des questions de droits d'accès aux classes publiques, lesquels, rappelons-le, ne se manifestent de toute façon que si toutes les classes n'appartiennent pas au même paquetage.

3 Les principes de mise en oeuvre d'un programme comportant plusieurs classes réparties dans différents fichiers source s'appliquent encore en cas de classes dérivées. Bien entendu, pour compiler une classe dérivée, il est nécessaire que sa classe de base appartienne au même fichier ou qu'elle ait déjà été compilée.

2 Accès d'une classe dérivée aux membres de sa classe de base

En introduction, nous avons dit qu'une classe dérivée hérite des champs et méthodes de sa classe de base. Mais nous n'avons pas précisé l'usage qu'elle peut en faire. Voyons précisément ce qu'il en est en distinguant les membres privés des membres publics.

2.1 Une classe dérivée n'accède pas aux membres privés

Dans l'exemple précédent, nous avons vu comment les membres publics de la classe de base restent des membres publics de la classe dérivée. C'est ainsi que nous avons pu appliquer la méthode *intialise* à un objet de type *Pointcol*.

En revanche, nous n'avons rien dit de la façon dont une méthode de la classe dérivée peut accéder aux membres de la classe de base. En fait, une classe dérivée n'a pas plus de droits d'accès à sa classe de base que n'importe quelle autre classe[1] :

> Une méthode d'une classe dérivée n'a pas accès aux membres privés de sa classe de base.

Cette règle peut paraître restrictive. Mais en son absence, il suffirait de créer une classe dérivée pour violer le principe d'encapsulation.

Si l'on considère la classe *Pointcol* précédente, elle ne dispose pour l'instant que d'une méthode *affiche*, héritée de *Point* qui, bien entendu, ne fournit pas la couleur. On peut chercher à la doter d'une nouvelle méthode nommée par exemple *affichec*, fournissant à la fois les coordonnées du point coloré et sa couleur. Il ne sera pas possible de procéder ainsi :

```
void affichec()  // méthode affichant les coordonnees et la couleur
{ System.out.println ("Je suis en " + x + " " + y) ; // NON : x et y sont privés
  System.out.println ("  et ma couleur est : " + couleur) ;
}
```

En effet, la méthode *affichec* de *Pointcol* n'a pas accès aux champs privés *x* et *y* de sa classe de base.

2.2 Elle accède aux membres publics

Comme on peut s'y attendre[2] :

> Une méthode d'une classe dérivée a accès aux membres publics de sa classe de base.

Ainsi, pour écrire la méthode *affichec*, nous pouvons nous appuyer sur la méthode *affiche* de *Point* en procédant ainsi :

```
public void affichec ()
{ affiche() ;
  System.out.println ("  et ma couleur est : " + couleur) ;
}
```

Une méthode d'affichage d'un objet de type Pointcol

1. Certains disent alors que la classe dérivée n'hérite pas des méthodes privées de sa classe de base. Mais cette terminologie nous semble ambiguë ; en effet, elle pourrait laisser entendre que ces membres n'appartiennent plus à la classe dérivée, alors qu'ils sont toujours présents (et accessibles par des méthodes publiques de la classe de base...).

2. Attention : ne confondez pas l'accès d'un objet à ses membres avec l'accès d'une classe (c'est-à-dire des ses méthodes) à ses membres (ou à ceux de sa classe de base), même si, ici, les droits sont identiques.

On notera que l'appel *affiche()* dans la méthode *affichec* est en fait équivalent à :

```
this.affiche() ;
```

Autrement dit, il applique la méthode *affiche* à l'objet (de type *Pointcol*) ayant appelé la méthode *affichec*.

Nous pouvons procéder de même pour définir dans *Pointcol* une nouvelle méthode d'initialisation nommée *initialisec*, chargée d'attribuer les coordonnées et la couleur à un point coloré :

```
public void initialisec (int x, int y, byte couleur)
{ initialise (x, y) ;
  this.couleur = couleur ;
}
```

Une méthode d'initialisation d'un objet de type Pointcol

2.3 Exemple de programme complet

Voici un exemple complet de programme reprenant cette nouvelle définition de la classe *Pointcol* et un exemple d'utilisation :

```
class Point
{ public void initialise (int abs, int ord)
  { x = abs ; y = ord ;
  }
  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }
  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}
class Pointcol extends Point
{ public void colore (byte couleur)
  { this.couleur = couleur ;
  }
  public void affichec ()
  { affiche() ;
    System.out.println ("  et ma couleur est : " + couleur) ;
  }
  public void initialisec (int x, int y, byte couleur)
  { initialise (x, y) ;
    this.couleur = couleur ;
  }
  private byte couleur ;
}
```

```
public class TstPcol2
{ public static void main (String args[])
  { Pointcol pc1 = new Pointcol() ;
    pc1.initialise (3, 5) ;
    pc1.colore ((byte)3) ;
    pc1.affiche() ;   // attention, ici affiche
    pc1.affichec() ;  // et ici affichec
    Pointcol pc2 = new Pointcol() ;
    pc2.initialisec(5, 8, (byte)2) ;
    pc2.affichec() ;
    pc2.deplace (1, -3) ;
    pc2.affichec() ;
  }
}
```

```
Je suis en 3 5
Je suis en 3 5
  et ma couleur est : 3
Je suis en 5 8
  et ma couleur est : 2
Je suis en 6 5
  et ma couleur est : 2
```

Une nouvelle classe Pointcol *et son utilisation*

Remarque

A propos des classes, il nous est arrivé de commettre un abus de langage qui consiste à parler :

– d'accès depuis l'extérieur d'une classe pour parler de l'accès d'un objet de la classe à ses membres,

– d'accès depuis l'intérieur d'une classe (ou même simplement d'accès d'une classe) pour parler de l'accès des méthodes de la classe à ses membres.

De même, il nous arrivera de parler de l'accès d'une classe dérivée aux membres de sa classe de base sans préciser qu'il s'agit de l'accès de ses méthodes.

Notons d'emblée que dans une classe dérivée :

– les membres publics de sa classe de base se comportent comme des membres publics de la classe dérivée,

– les membres privés de la classe de base se comportent comme des membres privés depuis l'extérieur de la classe dérivée ; en revanche, comme ils ne sont pas accessibles à la classe dérivée, on ne peut pas dire qu'ils se comportent comme de propres membres privés de la classe dérivée.

En C++

L'héritage existe, mais il comporte plus de possibilités qu'en Java. Tout d'abord, lors de la définition d'une classe dérivée, on peut restreindre ses droits d'accès aux membres de sa classe de base (on dispose finalement de trois sortes de dérivation : publique, privée ou protégée). Par ailleurs, contrairement à Java, C++ permet l'héritage multiple.

3 Construction et initialisation des objets dérivés

Dans les exemples précédents, nous avons volontairement choisi des classes sans constructeurs. En pratique, la plupart des classes en disposeront. Nous allons examiner ici les différentes situations possibles (présence ou absence de constructeur dans la classe de base et dans la classe dérivée). Puis nous préciserons, comme nous l'avons fait pour les classes simples (non dérivées), la chronologie des différentes opérations d'initialisation (implicites ou explicites) et d'appel des constructeurs.

3.1 Appels des constructeurs

Rappelons que dans le cas d'une classe simple (non dérivée), la création d'un objet par *new* entraîne l'appel d'un constructeur ayant la signature voulue[1] (nombre et type des arguments). Si aucun constructeur ne convient, on obtient une erreur de compilation sauf si la classe ne dispose d'aucun constructeur et que l'appel de *new* s'est fait sans argument. On a affaire alors à un pseudo-constructeur par défaut (qui ne fait rien). Voyons ce que deviennent ces règles avec une classe dérivée.

3.1.1 Exemple introductif

Examinons d'abord un exemple simple dans lequel la classe de base (*Point*) et la classe dérivée (*Pointcol*) disposent toutes les deux d'un constructeur (ce qui correspond à la situation la plus courante en pratique).

Pour fixer les idées, supposons que nous utilisions le schéma suivant, dans lequel la classe *Point* dispose d'un constructeur à deux arguments et la classe *Pointcol* d'un constructeur à trois arguments :

```
class Point
{ .....
  public Point (int x, int y)
  { .....
  }
  private int x, y ;
}
```

1. Moyennant d'éventuelles conversions légales.

```
class Pointcol extends Point
{ .....
  public Pointcol (int x, int y, byte couleur)
  { .....
  }
  private byte couleur ;
}
```

Tout d'abord, il faut savoir que :

> En Java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.

S'il est nécessaire d'initialiser certains champs de la classe de base et qu'ils sont convenablement encapsulés, il faudra disposer de fonctions d'altération ou bien recourir à un constructeur de la classe de base.

Ainsi, le constructeur de *Pointcol* pourrait :

- initialiser le champ *couleur* (accessible, car membre de *Pointcol*),
- appeler le constructeur de *Point* pour initialiser les champs *x* et *y*.

Pour ce faire, il est toutefois impératif de respecter une règle imposée par Java :

> Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il doit obligatoirement s'agir de la première instruction du constructeur et ce dernier est désigné par le mot clé **super**.

Dans notre cas, voici ce que pourrait être cette instruction :

```
super (x, y) ;   // appel d'un constructeur de la classe de base, auquel
                 // on fournit en arguments les valeurs de x et de y
```

D'où notre constructeur de *Pointcol* :

```
public Pointcol (int x, int y, byte couleur)
{ super (x, y) ;              // obligatoirement comme première instruction
  this.couleur = couleur ;
}
```

Voici un petit programme complet illustrant cette possibilité. Il s'agit d'une adaptation du programme du paragraphe 2.3, dans lequel nous avons remplacé les méthodes d'initialisation des classes *Point* et *Pointcol* par des constructeurs.

```
class Point
{ public Point (int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }
```

```
 public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}

class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;              // obligatoirement comme première instruction
    this.couleur = couleur ;
  }
  public void affichec ()
  { affiche() ;
    System.out.println ("  et ma couleur est : " + couleur) ;
  }
  private byte couleur ;
}

public class TstPcol3
{ public static void main (String args[])
  { Pointcol pc1 = new Pointcol(3, 5, (byte)3) ;
    pc1.affiche() ;   // attention, ici affiche
    pc1.affichec() ;  // et ici affichec

    Pointcol pc2 = new Pointcol(5, 8, (byte)2) ;
    pc2.affichec() ;
    pc2.deplace (1, -3) ;
    pc2.affichec() ;
  }
}
```

```
Je suis en 3 5
Je suis en 3 5
  et ma couleur est : 3
Je suis en 5 8
  et ma couleur est : 2
Je suis en 6 5
  et ma couleur est : 2
```

Appel du constructeur d'une classe de base dans un constructeur d'une classe dérivée

Remarques

1 Nous avons déjà signalé qu'il est possible d'appeler dans un constructeur un autre constructeur de la même classe, en utilisant le mot clé *this* comme nom de méthode. Comme celui effectué par *super*, cet appel doit correspondre à la première instruction du construc-

teur. Dans ces conditions, on voit qu'il n'est pas possible d'exploiter les deux possibilités en même temps.

2 Nous montrerons plus loin qu'une classe peut dériver d'une classe qui dérive elle-même d'une autre. L'appel par *super* ne concerne que le constructeur de la classe de base du niveau immédiatement supérieur.

3 Le mot clé *super* possède d'autres utilisations que nous examinerons au paragraphe 6.8.

En C++

C++ dispose d'un mécanisme permettant d'expliciter directement l'appel d'un constructeur d'une classe de base dans l'en-tête même du constructeur de la classe dérivée.

3.1.2 Cas général

L'exemple précédent correspond à la situation la plus usuelle, dans laquelle la classe de base et la classe dérivée disposent toutes les deux d'au moins un constructeur public (les constructeurs peuvent être surdéfinis sans qu'aucun problème particulier ne se pose). Dans ce cas, nous avons vu que le constructeur concerné de la classe dérivée doit prendre en charge l'intégralité de l'objet dérivé, quitte à s'appuyer sur un appel explicite du constructeur de la classe de base. Examinons les autres cas possibles, en distinguant deux situations :

- la classe de base ne possède aucun constructeur,
- la classe dérivée ne possède aucun constructeur.

La classe de base ne possède aucun constructeur

Il reste possible d'appeler le constructeur par défaut dans la classe dérivée, comme dans :

```
class A
{ .....   // aucun constructeur
}
class B extends A
{ public B (...)  // constructeur de B
  { super() ;     // appelle ici le pseudo-constructeur par défaut de A
    .....
  }
}
```

L'appel *super()* est ici superflu, mais il ne nuit pas. En fait, cette possibilité s'avère pratique lorsque l'on définit une classe dérivée sans connaître les détails de la classe de base. On peut ainsi s'assurer qu'un constructeur sans argument de la classe de base sera toujours appelé et, si cette dernière est bien conçue, que la partie de *B* héritée de *A* sera donc convenablement initialisée.

Bien entendu, il reste permis de ne doter la classe *B* d'aucun constructeur. Nous verrons qu'il y aura alors appel d'un constructeur par défaut de *A* ; comme ce dernier ne fait rien, au bout du compte, il ne se passera rien de particulier !

La classe dérivée ne possède aucun constructeur

Si la classe dérivée ne possède pas de constructeur, il n'est bien sûr plus question de prévoir un appel explicite (par *super*) d'un quelconque constructeur de la classe de base. On sait déjà que, dans ce cas, tout se passe comme s'il y avait appel d'un constructeur par défaut sans argument. Dans le cas d'une classe simple, ce constructeur par défaut ne faisait rien (nous l'avons d'ailleurs qualifié de "pseudo-constructeur"). Dans le cas d'une classe dérivée, il est prévu qu'il appelle un constructeur sans argument de la classe de base. On va retrouver ici les règles correspondant à la création d'un objet sans argument, ce qui signifie que la classe de base devra :

- soit posséder un constructeur public sans argument, lequel sera alors appelé,
- soit ne posséder aucun constructeur ; il y aura appel du pseudo-constructeur par défaut.

Voici quelques exemples.

Exemple 1

```
class A
{ public A() { ..... }        // constructeur 1 de A
  public A (int n) { ..... }  // constructeur 2 de A
}
class B extends A
{ ..... // pas de constructeur
}
B b = new B() ;    // construction de B --> appel de constructeur 1 de A
```

La construction d'un objet de type *B* entraîne l'appel du constructeur sans argument de *A*.

Exemple 2

```
class A
{ public A(int n) { ..... }  // constructeur 2 seulement
}
class B extends A
{ ..... // pas de constructeur
}
```

Ici, on obtient une erreur de compilation car le constructeur par défaut de *B* cherche à appeler un constructeur sans argument de *A*. Comme cette dernière dispose d'au moins un constructeur, il n'est plus question d'utiliser le constructeur par défaut de *A*.

Exemple 3

```
class A
{ ..... // pas de constructeur
}
class B extends A
{ ..... // pas de constructeur
}
```

Cet exemple ressemble au précédent, avec cette différence que *A* ne possède plus de constructeur. Aucun problème ne se pose plus. La création d'un objet de type *B* entraîne l'appel du constructeur par défaut de *B*, qui appelle le constructeur par défaut de *A*.

3.2 Initialisation d'un objet dérivé

Jusqu'ici, nous n'avons considéré que les constructeurs impliqués dans la création d'un objet dérivé. Mais comme nous l'avons déjà signalé au paragraphe 2.4 du chapitre 6 pour les classes simples, la création d'un objet fait intervenir plusieurs phases :

- allocation mémoire,
- initialisation par défaut des champs,
- initialisation explicite des champs,
- exécution des instructions du constructeur.

La généralisation à un objet d'une classe dérivée est assez intuitive. Supposons que :

```
class B extends A { ..... }
```

La création d'un objet de type *B* se déroule en 6 étapes.

1. Allocation mémoire pour un objet de type *B* ; il s'agit bien de l'intégralité de la mémoire nécessaire pour un tel objet, et pas seulement pour les champs propres à *B* (c'est-à-dire non hérités de *A*).
2. Initialisation par défaut de tous les champs de *B* (aussi bien ceux hérités de *A*, que ceux propres à *B*) aux valeurs "nulles" habituelles.
3. Initialisation explicite, s'il y a lieu, des champs hérités de *A* ; éventuellement, exécution des blocs d'initialisation de *A*.
4. Exécution du corps du constructeur de *A*.
5. Initialisation explicite, s'il y a lieu, des champs propres à *B* ; éventuellement, exécution des blocs d'initialisation de *B*.
6. Exécution du corps du constructeur de *B*.

Comme il a déjà été dit pour les classes simples (voir paragraphe 2.4.3 du chapitre 6), on aura intérêt en pratique à s'arranger pour que l'utilisateur de la classe n'ait pas besoin de s'interroger sur l'ordre chronologique exact de ces différentes opérations.

Remarque

Le constructeur à appeler dans la classe de base est souvent défini par la première instruction *super(...)* d'un constructeur de la classe dérivée. Cela pourrait laisser croire que l'initialisation explicite de la classe dérivée est faite avant l'appel du constructeur de la classe de base, ce qui n'est pas le cas. C'est d'ailleurs probablement pour cette raison que Java

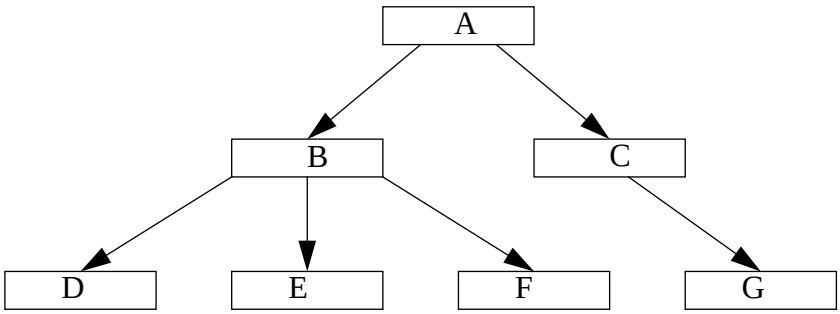
impose que cette instruction *super(...)* soit la première du constructeur : le compilateur est bien en mesure alors de définir le constructeur à appeler.

4 Dérivations successives

Jusqu'ici, nous n'avons raisonné que sur deux classes à la fois et nous parlions généralement de classe de base et de classe dérivée. En fait, comme on peut s'y attendre :

- d'une même classe peuvent être dérivées plusieurs classes différentes,
- les notions de classe de base et de classe dérivée sont relatives puisqu'une classe dérivée peut, à son tour, servir de classe de base pour une autre.

Autrement dit, on peut très bien rencontrer des situations telles que celle représentée par l'arborescence suivante[1] :



D est dérivée de *B*, elle-même dérivée de *A*. On dit souvent que *D* dérive de *A*. On dit aussi que *D* est une sous-classe de *A* ou que *A* est une super-classe de *D*. Parfois, on dit que *D* est une descendante de *A* ou encore que *A* est une ascendante de *D*. Naturellement, *D* est aussi une descendante de *B*. Lorsqu'on a besoin d'être plus précis, on dit alors que *D* est une descendante directe de *B*.

En C++

Contrairement à Java, C++ dispose de l'héritage multiple, notion généralement délicate à manipuler en Conception Orientée Objets. Nous verrons que Java dispose en revanche de la notion d'interface, qui offre des possibilités proches de celles de l'héritage multiple, sans en présenter les inconvénients.

1. Ici, les flèches vont de la classe de base vers la classe dérivée. On peut rencontrer la situation inverse.

5 Redéfinition et surdéfinition de membres

5.1 Introduction

Nous avons déjà étudié la notion de surdéfinition de méthode à l'intérieur d'une même classe. Nous avons vu qu'elle correspondait à des méthodes de même nom, mais de signatures différentes. Nous montrerons ici comment cette notion se généralise dans le cadre de l'héritage : une classe dérivée pourra à son tour surdéfinir une méthode d'une classe ascendante. Bien entendu, la ou les nouvelles méthodes ne deviendront utilisables que par la classe dérivée ou ses descendantes, mais pas par ses ascendantes.

Mais auparavant, nous vous présenterons la notion fondamentale de *redéfinition* d'une méthode. Une classe dérivée peut en effet fournir une nouvelle définition d'une méthode d'une classe ascendante. Cette fois, il s'agira non seulement de méthodes de même nom (comme pour la surdéfinition), mais aussi de même signature et de même type de valeur de retour. Alors que la surdéfinition permet de cumuler plusieurs méthodes de même nom, la redéfinition substitue une méthode à une autre.

Compte tenu de son importance, en particulier au niveau de ce que l'on nomme le polymorphisme, nous vous présenterons d'abord cette notion de redéfinition indépendamment des possibilités de surdéfinition. Nous verrons ensuite comment surdéfinition et redéfinition peuvent intervenir conjointement et nous vous livrerons les règles générales correspondantes.

5.2 La notion de redéfinition de méthod

Nous avons vu qu'un objet d'une classe dérivée peut accéder à toutes les méthodes publiques de sa classe de base. Considérons :

```
class Point
{ .....
  public void affiche()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}
class Pointcol extends Point
{ .....   // ici, on suppose qu'aucune méthode ne se nomme affiche
  private byte couleur ;
}
Point p ; Pointcol pc ;
```

L'appel *p.affiche()* fournit tout naturellement les coordonnées de l'objet *p* de type *Point*. L'appel *pc.affiche()* fournit également les coordonnées de l'objet *pc* de type *Pointcol*, mais bien entendu, il n'a aucune raison d'en fournir la couleur.

C'est la raison pour laquelle, dans l'exemple du paragraphe 2.3, nous avions introduit dans la classe *Pointcol* une méthode *affichec* affichant à la fois les coordonnées et la couleur d'un objet de type *Pointcol*.

Or, manifestement, ces deux méthodes *affiche* et *affichec* font un travail semblable : elles affichent les valeurs des données d'un objet de leur classe. Dans ces conditions, il paraît logique de chercher à leur attribuer le même nom.

Cette possibilité existe en Java ; elle se nomme *redéfinition de méthode*. Elle permet à une classe dérivée de redéfinir une méthode de sa classe de base, en en proposant une nouvelle définition. Encore faut-il respecter la signature de la méthode (type des arguments), ainsi que le type de la valeur de retour[1]. C'est alors cette nouvelle méthode qui sera appelée sur tout objet de la classe dérivée, *masquant* en quelque sorte la méthode de la classe de base.

Nous pouvons donc définir dans *Pointcol* une méthode *affiche* reprenant la définition actuelle de *affichec*. Si les coordonnées de la classe *Point* sont encapsulées et si cette dernière ne dispose pas de méthodes d'accès, nous devrons utiliser dans cette méthode *affiche* de *Pointcol* la méthode *affiche* de *Point*. Dans ce cas, un petit problème se pose ; en effet, nous pourrions être tentés d'écrire ainsi notre nouvelle méthode (en changeant simplement l'en-tête *affichec* en *affiche*) :

```
class Pointol extends Point
{ public void affiche()
  { affiche() ;
    System.out.println ("   et ma couleur est " + couleur) ;
  }
  .....
}
```

Or, l'appel *affiche()* provoquerait un appel récursif de la méthode *affiche* de *Pointcol*. Il faut donc préciser qu'on souhaite appeler non pas la méthode *affiche* de la classe *Pointcol*, mais la méthode *affiche* de sa classe de base. Il suffit pour cela d'utiliser le mot clé *super*, de cette façon :

```
public void affiche()
{ super.affiche() ;      // appel de la méthode affiche de la super-classe
  System.out.println ("   et ma couleur est " + couleur) ;
}
```

Dans ces conditions, l'appel *pc.affiche()* entraînera bien l'appel de *affiche* de *Pointcol*, laquelle, comme nous l'espérons, appellera *affiche* de *Point*, avant d'afficher la couleur.

Voici un exemple complet de programme illustrant cette possibilité :

```
class Point
{ public Point (int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;  }
```

1. Si la signature n'est pas respectée, on a affaire à une surdéfinition ; nous y reviendrons un peu plus loin. Le respect du type de la valeur de retour semble moins justifié ; on verra que Java l'impose pour faciliter l'utilisation du polymorphisme.

```
  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;              // obligatoirement comme première instruction
    this.couleur = couleur ;
  }
  public void affiche ()
  { super.affiche() ;
    System.out.println ("  et ma couleur est : " + couleur) ;
  }
  private byte couleur ;
}

public class TstPcol4
{ public static void main (String args[])
  { Pointcol pc = new Pointcol(3, 5, (byte)3) ;
    pc.affiche() ;  // ici, il s'agit de affiche de Pointcol
    pc.deplace (1, -3) ;
    pc.affiche() ;
  }
}
```

```
Je suis en 3 5
  et ma couleur est : 3
Je suis en 4 2
  et ma couleur est : 3
```

Exemple de redéfinition de la méthode affiche *dans une classe dérivée* Pointcol

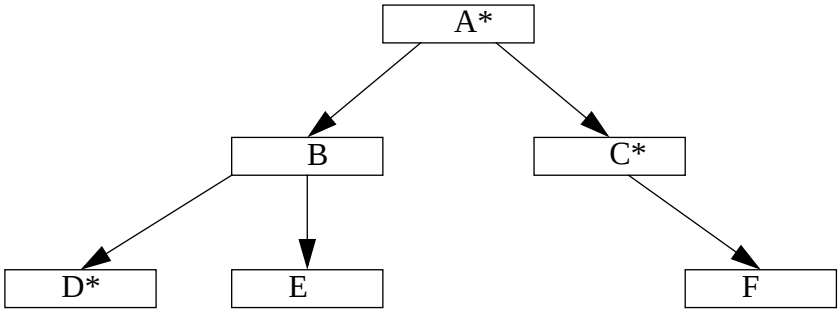
Remarque

Même si cela est fréquent, une redéfinition de méthode n'entraîne pas nécessairement comme ici l'appel par *super* de la méthode correspondante de la classe de base.

5.3 Redéfinition de méthode et dé ivations successives

Comme nous l'avons dit au paragraphe 4, on peut rencontrer des arborescences d'héritage aussi complexes qu'on le veut. Comme on peut s'y attendre, la redéfinition d'une méthode s'applique à une classe et à toutes ses descendantes jusqu'à ce qu'éventuellement l'une d'entre elles redéfinisse à nouveau la méthode. Considérons par exemple l'arborescence sui-

vante, dans laquelle la présence d'un astérisque (*) signale la définition ou la redéfinition d'une méthode *f* :



Dans ces conditions, l'appel de la méthode *f* conduira, pour chaque classe, à l'appel de la méthode indiquée en regard :

classe *A* : méthode *f* de *A*,
classe *B* : méthode *f* de *A*,
classe *C* : méthode *f* de *C*,
classe *D* : méthode *f* de *D*,
classe *E* : méthode *f* de *A*,
classe *F* : méthode *f* de *C*.

5.4 Surdéfinition et hé itage

Jusqu'à maintenant, nous n'avions considéré la surdéfinition qu'au sein d'une même classe. En Java, une classe dérivée peut surdéfinir une méthode d'une classe de base (ou, plus généralement, d'une classe ascendante). En voici un exemple :

```
class A
{ public void f (int n) { ..... }
  .....
}
class B extends A
{ public void f (float x) { ..... }
  .....
}
A a ; B b ;
int n ; float x ;
.....
a.f(n) ;  // appelle f (int) de A
a.f(x) ;  // erreur de compilation : une seule méthode acceptable (f(int) de A
          //   et on ne peut pas convertir x de float en int
b.f(n) ;  // appelle f (int) de A
b.f(x) ;  // appelle f(float) de B
```

Bien entendu, là encore, la recherche d'une méthode acceptable ne se fait qu'en remontant la hiérarchie d'héritage, jamais en la descendant... C'est pourquoi l'appel *a.f(x)* ne peut être satisfait, malgré la présence dans *B* d'une fonction *f* qui conviendrait.

En C++

La surdéfinition ne "franchit pas l'héritage". On considère un seul ensemble de méthodes d'une classe donnée (la première qui, en remontant la hiérarchie, possède au moins une méthode ayant le nom voulu).

5.5 Utilisation simultanée de surdéfinition et de redéfiniti

Surdéfinition et redéfinition peuvent cohabiter. Voyez cet exemple :

```
class A
{ .....
  public void f (int n)   { ..... }
  public void f (float x) { ..... }
}
class B extends A
{ .....
  public void f (int n)    { ..... }   // redéfinition de f(int) de A
  public void f (double y) { ..... }   // surdéfinition de f (de A et de B)
}
A a ; B b ;
int n ; float x ; double y ;
  .....
a.f(n) ;  // appel de f(int)    de A (mise en jeu de surdéfinition dans A)
a.f(x) ;  // appel de f(float)  de A (mise en jeu de surdéfinition dans A)
a.f(y) ;  // erreur de compilation
b.f(n) ;  // appel de f(int)    de B (mise en jeu de redéfinition)
b.f(x) ;  // appel de f(float)  de A (mise en jeu de surdéfinition dans A et B)
b.f(y) ;  // appel de f(double) de B (mise en jeu de surdéfinition dans A et B)
```

Remarque

La richesse des possibilités de cohabitation entre surdéfinition et redéfinition peut conduire à des situations complexes qu'il est généralement préférable d'éviter en soignant la conception des classes.

5.6 Contraintes portant sur la redéfinitio

5.6.1 Valeur de retour

Lorsqu'on surdéfinit une méthode, on n'est pas obligé de respecter le type de la valeur de retour. Cet exemple est légal :

```
class A
{ .....
  public int f(int n) { .....}
}
class B extends A
{ .....
  public float f(float x) { ..... }
}
```

En revanche, en cas de redéfinition, Java impose non seulement l'identité des signatures, mais aussi celle du type de la valeur de retour :

```
class A
{ public int f (int n) { ..... }
}
class B extends A
{ public float f (int n) { ..... }     // erreur
}
```

Ici, on n'a pas affaire à une surdéfinition de *f* puisque la signature de la méthode est la même dans *A* et dans *B*. Il devrait donc s'agir d'une redéfinition, mais comme les types de retour sont différents, on aboutit à une erreur de compilation.

Remarque

Dores et déjà, on voit que ces règles apportent une certaine homogénéité dans la manipulation d'objets d'un type de base et d'objets d'un type dérivé. Les deux types d'objets peuvent se voir appliquer une méthode donnée de la même façon syntaxique ; cette remarque concerne aussi l'usage qui est fait de la valeur de retour (alors que la méthode effectivement appelée est différente). Ces règles prendront encore plus d'intérêt dans le contexte du polymorphisme dont nous parlons un peu plus loin.

En C++

D'après ce que nous venons de voir, la situation suivante est illégale en Java :

```
class A
{ public A f () { ..... }
}
class B extends A
{ public B f () { ..... }  // illégal : redéfinition de f avec type de
}                          // la valeur de retour différent
```

En C++, elle peut l'être (à condition que la fonction *f* soit "virtuelle" et que les valeurs de retour soient transmises par pointeur ou par référence). On a affaire à ce qu'on nomme parfois des méthodes virtuelles *covariantes*.

5.6.2 Les droits d'accès

Considérons cet exemple :

```
class A
{ public void f (int n) { ..... }
}
class B extends A
{ private void f (int n) { ..... }  // tentative de redéfinition de f de A
}
```

Il est rejeté par le compilateur. S'il était accepté, un objet de classe *A* aurait accès à la méthode *f*, alors qu'un objet de classe dérivée *B* n'y aurait plus accès. La classe dérivée romprait en quelque sorte le contrat établi par la classe A. C'est pourquoi **la redéfinition d'une méthode ne doit pas diminuer les droits d'accès à cette méthode**. En revanche, elle peut les augmenter, comme dans cet exemple :

```
class A
{ private void f (int n) { ..... }
}
class B extends A
{ public void f (int n) { ..... }  // redéfinition de f avec extension
}                                  // des droits d'accès
```

Ici, on redéfinit *f* dans la classe dérivée et, en plus, on la rend accessible à l'extérieur de la classe. On pourrait penser qu'on viole ainsi l'encapsulation des données. En fait, il n'en est rien puisque la méthode *f* de *B* n'a pas accès aux membres privés de *A*. Simplement, tout se passe comme si la classe *B* était dotée d'une fonctionnalité supplémentaire par rapport à *A*.

Rappelons que nous avons rencontré trois sortes de droits d'accès à une classe : *public*, *private* et "rien" correspondant à l'accès de paquetage (revoyez éventuellement le paragraphe 13.4.2 du chapitre 6). Nous verrons un peu plus loin qu'il en existe un quatrième, *protected*, d'ailleurs peu utilisé. Ces quatre droits d'accès se classent dans cet ordre, du plus élevé au moins élevé :

public absence de mention (droit de paquetage) *protected* *public*

On trouvera à l'Annexe A un récapitulatif de tout ce qui concerne les différents droits d'accès à une classe, un membre, une classe interne ou une interface.

5.7 Règles générales de redéfinition et de surdéfiniti

Compte tenu de la complexité de la situation,voici un récapitulatif des règles de surdéfinition et de redéfinition.

Si une méthode d'une classe dérivée a la même signature qu'une méthode d'une classe ascendante :

- les valeurs de retour des deux méthodes doivent être exactement de même type,
- le droit d'accès de la méthode de la classe dérivée ne doit pas être moins élevé que celui de la classe ascendante,
- la clause *throws* de la méthode de la classe dérivée ne doit pas mentionner des exceptions non mentionnées dans la clause *throws* de la méthode de la classe ascendante (la clause *throws* sera étudiée au Chapitre 10).

Si ces trois conditions sont remplies, on à affaire à une *redéfinition*. Sinon, il s'agit d'une erreur.

Dans les autres cas, c'est-à-dire lorsqu'une méthode d'une classe dérivée a le même nom qu'une méthode d'une classe ascendante, avec une signature différente, on a affaire à une *surdéfinition*. Cela est vrai, quels que soient les droits d'accès des deux méthodes. La nouvelle méthode devient utilisable par la classe dérivée et ses descendantes éventuelles (selon ses propres droits d'accès), sans masquer les méthodes de même nom des classes ascendantes.

Remarques

1 Une méthode de classe (*static*) ne peut pas être redéfinie dans une classe dérivée. Cette restriction va de soi puisque c'est le type de l'objet appelant une méthode qui permet de choisir entre la méthode de la classe de base et celle de la classe dérivée. Comme une méthode de classe peut être appelée sans être associée à un objet, on comprend qu'un tel choix ne soit plus possible.

2 Les possibilités de redéfinition d'une méthode prendront tout leur intérêt lorsqu'elles seront associées au polymorphisme que nous étudions un peu plus loin.

5.8 Duplication de champs

Bien que cela soit d'un usage peu courant, une classe dérivée peut définir un champ portant le même nom qu'un champ d'une classe de base ou d'une classe ascendante. Considérons cette situation, dans laquelle nous avons exceptionnellement prévu des champs publics :

```
class A
{ public int n ;
  .....
}
class B extends A
{ public float n ;
```

```
 public void f()
  { n = 5.25f ;   // n désigne le champ n (float) de B
    super.n = 3 ; // tandis que super.n désigne le champ n (int) de la super-
                  // classe de B
  }
}
A a ; B b ;
a.n = 5 ;     // a.n désigne ici le champ n(int) de la classe A
b.n = 3.5f ;  // b.n désigne ici le champ n(float) de la classe B
```

Il n'y a donc pas redéfinition du champ *n* comme il y a redéfinition de méthode, mais création d'un nouveau champ qui s'ajoute à l'ancien. Toutefois, alors que les deux champs peuvent encore être utilisés depuis la classe dérivée, seul le champ de la classe dérivée n'est visible de l'extérieur. Voici un petit programme complet très artificiel illustrant la situation :

```
class A
{ public int n= 4 ; }
class B extends A
{ public float n = 4.5f ; }
public class DupChamp
{ public static void main(String[]args)
  { A a = new A() ; B b = new B() ;
    System.out.println ("a.n = " + a.n) ;
    System.out.println ("b.n = " + b.n) ;
  }
}
```

```
a.n = 4
b.n = 4.5
```

Exemple (artificiel) de duplication de champs

6 Le polymorphisme

Voyons maintenant comment Java permet de mettre en oeuvre ce qu'on nomme généralement le *polymorphisme*. Il s'agit d'un concept extrêmement puissant en P.O.O., qui complète l'héritage. On peut caractériser le polymorphisme en disant qu'il permet de manipuler des objets sans en connaître (tout à fait) le type. Par exemple, on pourra construire un tableau d'objets (donc en fait de références à des objets), les uns étant de type *Point*, les autres étant de type *Pointcol* (dérivé de *Point*) et appeler la méthode *affiche* pour chacun des objets du tableau. Chaque objet réagira en fonction de son propre type.

Mais il ne s'agira pas de traiter ainsi n'importe quel objet. Nous montrerons que le polymorphisme exploite la relation *est* induite par l'héritage en appliquant la règle suivante : un point coloré est aussi un point, on peut donc bien le traiter comme un point, la réciproque étant bien sûr fausse.

6.1 Les bases du polymorphisme

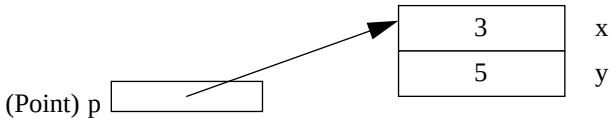
Considérons cette situation dans laquelle les classes *Point* et *Pointcol* sont censées disposer chacune d'une méthode *affiche*, ainsi que des constructeurs habituels (respectivement à deux et trois arguments) :

```
class Point
{ public Point (int x, int y) { ..... }
  public void affiche () { ..... }
}
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  public void affiche () { ..... }
}
```

Avec ces instructions :

```
Point p ;
p = new Point (3, 5) ;
```

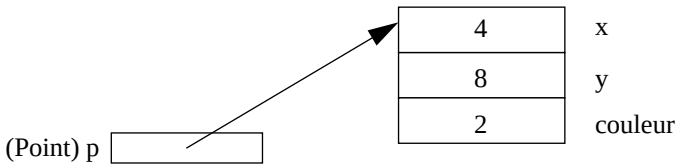
on aboutit tout naturellement à cette situation :



Mais il se trouve que Java autorise ce genre d'affectation (*p* étant toujours de type *Point*)

```
p = new Pointcol (4, 8, (byte)2) ;   // p de type Point contient la référence
                                     //   à un objet de type Pointcol
```

La situation correspondante est la suivante :



D'une manière générale, Java permet d'affecter à une variable objet non seulement la référence à un objet du type correspondant, mais aussi une référence à un objet d'un type dérivé. On peut dire qu'on est en présence d'une conversion implicite (légale) d'une référence à un type classe *T* en une référence à un type ascendant de *T* ; on parle aussi de compatibilité par affectation entre un type classe et un type ascendant.

Considérons maintenant ces instructions :

```
Point p = new Point (3, 5) ;
p.affiche () ;     // appelle la méthode affiche de la classe Point
```

```
p = new Poincol (4, 8, 2) ;
p.affiche () ;     // appelle la méthode affiche de la classe Pointcol
```

Dans la dernière instruction, la variable *p* est de type *Point*, alors que l'objet référencé par *p* est de type *Pointcol*. L'instruction *p.affiche()* appelle alors la méthode *affiche* de la classe *Pointcol*. Autrement dit, elle se fonde, non pas sur le type de la variable *p*, mais bel et bien sur le type effectif de l'objet référencé par *p* au moment de l'appel (ce type pouvant évoluer au fil de l'exécution). Ce choix d'une méthode au moment de l'exécution (et non plus de la compilation) porte généralement le nom de *ligature dynamique* (ou encore de liaison dynamique).

En résumé, le polymorphisme en Java se traduit par :

- la compatibilité par affectation entre un type classe et un type ascendant,
- la ligature dynamique des méthodes.

Le polymorphisme permet d'obtenir un comportement adapté à chaque type d'objet, sans avoir besoin de tester sa nature de quelque façon que ce soit. La richesse de cette technique amène parfois à dire que l'instruction *switch* est à la P.O.O. ce que l'instruction *goto* est à la programmation structurée. Autrement dit, le bon usage de la P.O.O (et du polymorphisme) permet parfois d'éviter des instructions de test, de même que le bon usage de la programmation structurée permettait d'éviter l'instruction *goto*.

En C++

En C++, on dispose des mêmes règles de compatibilité entre un type classe et un type ascendant. En revanche, le choix d'une méthode est, par défaut, réalisé à la compilation (on parle de ligature statique). Mais il est possible de déclarer certaines méthodes *virtuelles* (mot clé *virtual*), ce qui a pour effet de les soumettre à une ligature dynamique. En Java, tout se passe comme si toutes les méthodes étaient virtuelles.

Exemple 1

Voici un premier exemple intégrant les situations exposées ci-dessus dans un programme complet :

```
class Point
{ public Point (int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }
  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}
```

```
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;              // obligatoirement comme première instruction
    this.couleur = couleur ;
  }
  public void affiche ()
  { super.affiche() ;
    System.out.println ("  et ma couleur est : " + couleur) ;
  }
  private byte couleur ;
}
public class Poly
{ public static void main (String args[])
  { Point p = new Point (3, 5) ;
    p.affiche() ;  // appelle affiche de Point
    Pointcol pc = new Pointcol (4, 8, (byte)2) ;
    p = pc ;       // p de type Point, reference un objet de type Pointcol
    p.affiche() ;  // on appelle affiche de Pointcol
    p = new Point (5, 7) ;  // p reference a nouveau un objet de type Point
    p.affiche() ;  // on appelle affiche de Point
  }
}
```

```
Je suis en 3 5
Je suis en 4 8
  et ma couleur est : 2
Je suis en 5 7
```

Exemple de polymorphisme

Exemple 2

Voici un second exemple de programme complet dans lequel nous exploitons les possibilités de polymorphisme pour créer un tableau "hétérogène" d'objets, c'est-à-dire dans lequel les éléments peuvent être de type différent.

```
class Point
{ public Point (int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;              // obligatoirement comme première instruction
    this.couleur = couleur ;
  }
```

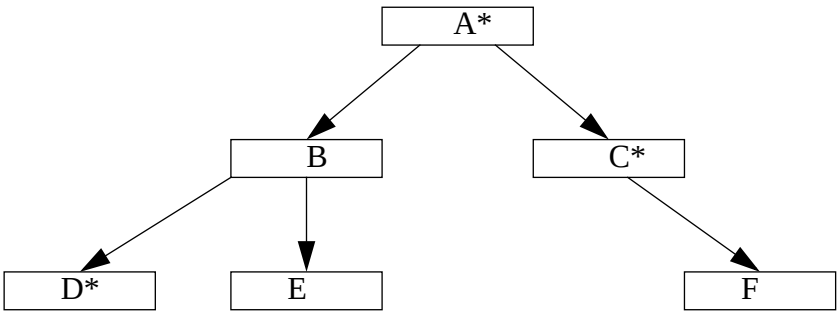
```
  public void affiche ()
  { super.affiche() ;
    System.out.println ("  et ma couleur est : " + couleur) ;
  }
  private byte couleur ;
}
public class TabHeter
{ public static void main (String args[])
  { Point [] tabPts = new Point [4] ;
    tabPts [0] = new Point (0, 2) ;
    tabPts [1] = new Pointcol (1, 5, (byte)3) ;
    tabPts [2] = new Pointcol (2, 8, (byte)9) ;
    tabPts [3] = new Point (1, 2) ;
    for (int i=0 ; i< tabPts.length ; i++) tabPts[i].affiche() ;
  }
}
```

```
Je suis en 0 2
Je suis en 1 5
  et ma couleur est : 3
Je suis en 2 8
  et ma couleur est : 9
Je suis en 1 2
```

Exemple d'utilisation du polymorphisme pour gérer un tableau hétérogène

6.2 Généralisation à plusieurs classes

Nous venons de vous exposer les fondements du polymorphisme en ne considérant que deux classes. Mais il va de soi qu'ils se généralisent à une hiérarchie quelconque. Considérons à nouveau la hiérarchie de classes présentée au paragraphe 5.3, dans laquelle seules les classes marquées d'un astérisque définissent ou redéfinissent la méthode *f* :



Avec ces déclarations :

```
A a ; B b ; C c ; D d ; E e ; F f ;
```

les affectations suivantes sont légales :

```
a = b ; a = c ; a = d ; a = e ; a =f ;
b = d ; b = e ;
c = f ;
```

En revanche, celles-ci ne le sont pas :

```
b = a ;   // erreur : A ne descend pas de B
d = c ;   // erreur : C ne descend pas de D
c = d ;   // erreur : D ne descend pas de C
```

Voici quelques exemples précisant la méthode *f* appelée, selon la nature de l'objet effectivement référencé par *a* (de type *A*) :

a référence un objet de type *A* : méthode *f* de *A*

a référence un objet de type *B* : méthode *f* de *A*

a référence un objet de type *C* : méthode *f* de *C*

a référence un objet de type *D* : méthode *f* de *D*

a référence un objet de type *E* : méthode *f* de *A*

a référence un objet de type *F* : méthode *f* de *C*.

Remarque

Cet exemple est très proche de celui présenté au paragraphe 5.2 à propos de la redéfinition d'une méthode. Mais ici, la variable contenant la référence à un objet de l'une des classes est toujours de type *A*, alors que, auparavant, elle était du type de l'objet référencé. L'exemple du paragraphe 6.3 peut en fait être considéré comme un cas particulier de celui-ci.

6.3 Autre situation où l'on exploite le polymorphisme

Dans les exemples précédents, les méthodes *affiche* de *Point* et de *Pointcol* se contentaient d'afficher les valeurs des champs concernés, sans préciser la nature exacte de l'objet. Nous pourrions par exemple souhaiter que l'information se présente ainsi pour un objet de type *Point* :

```
Je suis un point
  Mes coordonnees sont : 0 2
```

et ainsi pour un objet de type *Pointcol* :

```
Je suis un point colore de couleur 3
  Mes coordonnees sont : 1 5
```

On peut considérer que l'information affichée par chaque classe se décompose en deux parties : une première partie spécifique à la classe dérivée (ici *Pointcol*), une seconde partie commune correspondant à la partie héritée de *Point* : les valeurs des coordonnées. D'une

manière générale, ce point de vue pourrait s'appliquer à toute classe descendant de *Point*. Dans ces conditions, plutôt que de laisser chaque classe descendant de *Point* redéfnir la méthode *affiche*, on peut définir la méthode *affiche* de la classe *point* de manière qu'elle :

- affiche les coordonnées (action commune à toutes les classes),
- fasse appel à une autre méthode (nommée par exemple *identifie*) ayant pour vocation d'afficher les informations spécifiques à chaque objet. Ce faisant, nous supposons que chaque descendante de *Point* redéfinira *identifie* de façon appropriée (mais elle n'aura plus à prendre en charge l'affichage des coordonnées).

Cette démarche nous conduit à définir la classe *Point* de la façon suivante :

```
class Point
{ public Point (int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void affiche ()
  { identifie() ;
    System.out.println ("  Mes coordonnees sont : " + x + " " + y) ;
  }

public void identifie ()
  { System.out.println ("Je suis un point ") ;
  }
  private int x, y ;
}
```

Dérivons une classe *Pointcol* en redéfinissant comme voulu la méthode *identifie* :

```
class Pointcol extends Point
{
  public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;
    this.couleur = couleur ;
  }

  public void identifie ()
  { System.out.println ("Je suis un point colore de couleur " + couleur) ;
  }
  private byte couleur ;
}
```

Considérons alors ces instructions :

```
Pointcol pc = new Pointcol (8, 6, (byte)2) ;
pc.affiche () ;
```

L'instruction *pc.affiche()* entraîne l'appel de la méthode *affiche* de la classe *Point* (puisque cette méthode n'a pas été redéfinie dans *Pointcol*). Mais dans la méthode *affiche* de *Point*,

l'instruction *identifie()* appelle la méthode *identifie* de la classe correspondant à l'objet effectivement concerné (autrement dit, celui de référence *this*). Comme ici, il s'agit d'un objet de type *Pointcol*, il y aura bien appel de la méthode *identifie* de *Pointcol*.

La même analyse s'appliquerait à la situation :

```
Point p
p = new Pointcol (8, 6, (byte)2) ;
p.affiche () ;
```

Là encore, c'est le type de l'objet référencé par *p* qui interviendra dans le choix de la méthode *affiche*.

Voici un programme complet reprenant les définitions des classes *Point* et *Pointcol* utilisant la même méthode *main* que l'exemple du paragraphe 6.1 pour gérer un tableau hétérogène :

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

  public void affiche ()
  { identifie() ;
    System.out.println ("  Mes coordonnees sont : " + x + " " + y) ;
  }
  public void identifie ()
  { System.out.println ("Je suis un point ") ;
  }
  private int x, y ;
}
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;
    this.couleur = couleur ;
  }
  public void identifie ()
  { System.out.println ("Je suis un point colore de couleur " + couleur) ;
  }
  private byte couleur ;
}
public class TabHet2
{ public static void main (String args[])
  { Point [] tabPts = new Point [4] ;
    tabPts [0] = new Point (0, 2) ;
    tabPts [1] = new Pointcol (1, 5, (byte)3) ;
    tabPts [2] = new Pointcol (2, 8, (byte)9) ;
    tabPts [3] = new Point (1, 2) ;
    for (int i=0 ; i< tabPts.length ; i++)
      tabPts[i].affiche() ;
  }
}
```

```
Je suis un point
  Mes coordonnees sont : 0 2
Je suis un point colore de couleur 3
  Mes coordonnees sont : 1 5
Je suis un point colore de couleur 9
  Mes coordonnees sont : 2 8
Je suis un point
  Mes coordonnees sont : 1 2
```

Une autre situation où le polymorphisme se révèle indispensable

Remarque

La technique proposée ici s'applique à n'importe quel objet d'une classe descendant de *Point*, pour peu qu'elle redéfinisse correctement la méthode *identifie*. Le code de *affiche* de *Point* n'a pas besoin de connaître ce que sont ou seront les descendants de la classe, ce qui ne l'empêche pas de les manipuler.

6.4 Polymorphisme, redéfinition et surdéfiniti

Par essence, le polymorphisme se fonde sur la redéfinition des méthodes. Mais il est aussi possible de surdéfinir une méthode ; nous avons exposé au paragraphe 5.5 les règles concernant ces deux possibilités. Cependant, nous n'avions pas tenu compte alors des possibilités de polymorphisme qui peuvent conduire à des situations assez complexes. En voici un exemple :

```
class A
{ public void f (float x) { ..... }
  .....
}
class B extends A
{ public void f (float x) { ..... }   // redéfinition de f de A
  public void f (int n)   { ..... }   // surdéfinition de f pour A et B
  .....
}
A a = new A(...) ;
B b = new B(...) ; int n ;
a.f(n) ;    // appelle f (float) de A (ce qui est logique)
b.f(n) ;    // appelle f(int) de B comme on s'y attend
a = b  ;    // a contient une référence sur un objet de type B
a.f(n) ;    // appelle f(float) de B
```

Ainsi, bien que les instructions *b.f(n)* et *a.f(n)* appliquent toutes les deux une méthode *f* à un objet de type *B*, elles n'appellent pas la même méthode. Voyons plus précisément pourquoi. En présence de l'appel *a.f(n)*, le compilateur recherche la meilleure méthode (règles de surdéfinition) parmi toutes les méthodes de la classe correspondant au type de *a* (ici *A*) ou ses

ascendantes. Ici, il s'agit de *void f(float x)* de *A*. A ce stade donc, la signature de la méthode et son type de retour sont entièrement figés. Lors de l'exécution, on se fonde sur le type de l'objet référencé par *a* pour rechercher une méthode ayant la signature et le type de retour voulus. On aboutit alors à la méthode *void f (float x)* de *A*, et ce malgré la présence dans *B* d'une méthode qui serait mieux adaptée au type de l'argument effectif.

Ainsi, malgré son aspect ligature dynamique, le polymorphisme se fonde sur une signature et un type de retour définis à la compilation (et qui ne seront donc pas remis en question lors de l'exécution).

6.5 Conversions des arguments effectifs

Nous avons déjà vu comment, lors de l'appel d'une méthode, ses arguments effectifs pouvaient être soumis à des conversions implicites. Ces dernières peuvent également intervenir dans la recherche d'une fonction surdéfinie.

Toutefois, nos exemples se limitaient jusqu'ici à des conversions implicites de types primitifs. Or, la conversion d'un type dérivé en un type de base est aussi une conversion implicite légale ; elle va donc pouvoir être utilisée pour les arguments effectifs d'une méthode. Nous en allons en voir ici des exemples, d'abord dans des situations d'appel simple, puis dans des situations de surdéfinition. Aucune information nouvelle ne sera apportée ; nous nous contenterons d'exploiter les règles déjà rencontrées.

6.5.1 Cas d'une méthode non surdéfinie

Considérons tout d'abord cette situation (nous utilisons une méthode statique *f* d'une classe *Util* par souci de simplification) :

```
class A
{ public void identite()
  { System.out.println ("objet de type A") ;
  }
}
class B extends A
{    // pas de redéfinition de identite ici
}
class Util
{ static void f(A a)       // f attend un argument de type A
  { a.identite() ;
  }
}
   .....
A a = new A() ; B b = new B() ;
Util.f(a) ;    // OK : appel usuel ; il affiche : "objet de type A"
Util.f(b) ;    // OK : une référence à un objet de type B
               //  est compatible avec une référence à un objet de type A
               //      l'appel a.identite affiche : "objet de type A"
```

Si, en revanche, nous modifions ainsi la définition de B :

```
class B extends A
{  public void identite ()
   { System.out.println ("objet de type B") ;    // redéfinition de identite
   }
}

f(b) ;         // OK : une référence à un objet de type B
               //      est compatible avec une référence à un objet de type A
               //      mais cet appel affiche : "objet de type B"
```

L'appel *f(b)* entraîne toujours la conversion de *b* en *A*. Mais dans *f*, l'appel *a.identite()* conduit à l'appel de la méthode *identite* définie par le type de l'objet réellement référencé par *a* et l'on obtient bien l'affichage de "objet de type B".

Cet exemple montre que, comme dans les situations d'affectation, la conversion implicite d'un type dérivé dans un type de base n'est pas dégradante puisque, grâce au polymorphisme, c'est bien le type de l'objet référencé qui interviendra.

6.5.2 Cas d'une méthode surdéfinie

Voici un premier exemple relativement naturel (là encore, l'utilisation de méthodes *f* statiques n'est destiné qu'à simplifier les choses) :

```
class A { ..... }
class B extends A { ..... }
class Util
{ static void f(int p, B b)   { ..... }
  static void f(float x, A a) { ..... }
}
   .....
A a = new A() ; B b = new B() ;
int n ; float x ;
Util.f(n, b) ;     // OK sans conversions :      appel de f(int, B)
Util.f(x, a) ;     // OK sans conversions :      appel de f(float, A)
Util.f(n, a) ;     // conversion de n en float : appel de f(float, A)
Util.f(x, b) ;     // conversion de b en A :     appel de f(float, A)
```

Voici un second exemple, un peu moins trivial :

```
class A { ..... }
class B extends A { ..... }
class Util
{ static void f(int p, A a)
  { ..... }
  static void f(float x, B b)
  { ..... }
}
   .....
A a = new A() ; B b = new B() ;
int n ; float x ;
Util.f(n, a) ;   // OK sans conversions :  appel de f(int, A)
```

```
Util.f(x, b) ;   // OK sans conversions :  appel de f(float, B)
Util.f(n, b) ;   // erreur compilation car ambigu : deux possibilités :
                 //    soit convertir n en float et utiliser f(float, B)
                 //    soit convertir b en A et utiliser f(int, A)
Util.f(x, a ) ;  // erreur compilation : aucune fonction ne convient
                 //   (on ne peut pas convertir implicitement de A en B
                 //    ni de float en int)
```

6.6 Les règles du polymorphisme en Java

Dans les situations usuelles, le polymorphisme est facile à comprendre et à exploiter. Cependant, nous avons vu que l'abus des possibilités de surdéfinition des méthodes pouvait conduire à des situations complexes. Aussi, nous vous proposons ici de récapituler les différentes règles rencontrées progressivement dans ce paragraphe 6.

Compatibilité. Il existe une conversion implicite d'une référence à un objet de classe *T* en une référence à un objet d'une classe ascendante de *T* (elle peut intervenir aussi bien dans les affectations que dans les arguments effectifs).

Ligature dynamique. Dans un appel de la forme *x.f(...)* où *x* est supposé de classe *T*, le choix de *f* est déterminé ainsi :

- *à la compilation* : on détermine dans la classe *T* ou ses ascendantes la signature de la meilleure méthode *f* convenant à l'appel, ce qui définit du même coup le type de la valeur de retour.

- *à l'exécution* : on recherche la méthode *f* de signature et de type de retour voulus, à partir de la classe correspondant au type effectif de l'objet référencé par *x* (il est obligatoirement de type *T* ou descendant) ; si cette classe ne comporte pas de méthode appropriée, on remonte dans la hiérarchie jusqu'à ce qu'on en trouve une (au pire, on remontera jusqu'à *T*).

6.7 Les conversions explicites de références

Nous avons largement insisté sur la compatibilité qui existe entre référence à un objet d'un type donné et référence à un objet d'un type ascendant. Comme on peut s'y attendre, la compatibilité n'a pas lieu dans le sens inverse. Considérons cet exemple, fondé sur nos classes *Point* et *Pointcol* habituelles :

```
class Point { ..... }
class Pointcol extends Point { ..... }
   .....
Pointcol pc ;
pc = new Point (...) ;    // erreur de compilation
```

Si l'affectation était légale, un simple appel tel que *pc.colore(...)* conduirait à attribuer une couleur à un objet de type *Point*, ce qui poserait quelques problèmes à l'exécution...

Mais considérons cette situation :

```
Point p ;
Pointcol pc1 = new Pointcol(...), pc2 ;
   .....
```

```
p = pc1 ;   // p contient la référence à un objet de type Pointcol
   .....
pc2 = p ;   // refusé en compilation
```

L'affectation *pc2 = p* est tout naturellement refusée. Cependant, nous sommes certains que *p* contient bien ici la référence à un objet de type *Pointcol*. En fait, nous pouvons forcer le compilateur à réaliser la conversion correspondante en utilisant l'opérateur de *cast* déjà rencontré pour les types primitifs. Ici, nous écrirons simplement :

```
pc2 = (Pointcol) p ;   // accepté en compilation
```

Toutefois, lors de l'exécution, Java s'assurera que *p* contient bien une référence à un objet de type *Pointcol* (ou dérivé) afin de ne pas compromettre la bonne exécution du programme. Dans le cas contraire, on obtiendra une exception *ClassCastException* qui, si elle n'est pas traitée (comme on apprendra à le faire au Chapitre 10), conduira à un arrêt de l'exécution.

Comme on peut s'y attendre, ce genre de conversion explicite n'est à utiliser qu'en toute connaissance de cause.

Remarque

On peut s'assurer qu'un objet est bien une instance d'une classe donnée en recourant à l'opérateur *instanceOf*. Par exemple, l'expression *p.instanceOf (Point)* vaudra *true* si *p* est (exactement) de type *Point*.

6.8 Le mot clé super

Nous avons vu à plusieurs reprises que le mot clé *super* permettait d'accéder à une méthode d'une classe de base, alors que cette dernière était redéfinie dans la classe dérivée. En général, vous pourrez vous contenter d'appliquer mécaniquement la démarche proposée. Mais pour être exhaustif, il faut préciser que ce mot clé est en fait une référence gérée de manière particulière. Considérons cet exemple de programme :

```
class A
{ void f()
  { System.out.println ("appel f de A") ;
  }
}
class B extends A
{ void f()
  { System.out.println ("appel f de B") ;
  }
  public void test()
  { A a = super ;
    a.f() ;
    super.f() ;
    this.f() ;
  }
}
```

```
public class Super
{ public static void main (String args[])
  { B b = new B() ;
    b.test() ;
  }
}
```

```
appel f de B
appel f de A
appel f de B
```

La référence super *n'est pas soumise à la ligature dynamique*

Dans la méthode *test* de la classe *B*, *super* est une référence à un objet de type *A*. Cependant, elle référence l'objet lui-même (de type *B*). Nous avons déjà vu que l'appel *super.f()* entraînait l'appel de la méthode *f* de l'ascendant *A*. En revanche, vous constatez que l'appel *a.f()* entraîne bien quant à lui l'appel de la méthode *f* de *B*, comme le laisse prévoir le polymorphisme.

Ainsi, bien qu'il corresponde à une référence, le mot clé *super* n'est pas traité comme une référence usuelle puisqu'il fait exception au polymorphisme.

6.9 Limites de l'héritage et du polymorphisme

La puissance des techniques d'héritage et de polymorphisme finit parfois par en faire oublier les règles exactes et les limitations qui en découlent.

Considérez la situation suivante dans laquelle :

- la classe *Point* dispose d'une méthode *identique* fournissant la valeur *true* lorsque le point fourni en argument a les mêmes coordonnées que le point courant :

```
Point p1, p2 ;
   .....
p1.identique(p2)   // true si p1 et p2 ont mêmes coordonnées
```

- la classe *Pointcol*, dérivée de *Point*, redéfinit cette méthode pour prendre en compte non seulement l'égalité des coordonnées, mais aussi celle de la couleur :

```
Pointcol pc1, pc2 ;
   .....
pc1.identique(pc2)   // true si pc1 et pc2 ont mêmes coordonnées et même couleur
```

Considérons alors :

```
Point p1 = new Pointcol (1, 2, (byte)5) ;
Point p2 = new Pointcol (1, 2, (byte)8) ;
```

L'expression *p1.identique(p2)* a pour valeur *true* alors que nos deux points colorés n'ont pas la même couleur. L'explication réside tout simplement dans la bonne application des règles relatives au polymorphisme. En effet, lors de la compilation de cette expression *p1.identique(p2)*, on s'est fondé sur le type de *p1* pour en déduire que l'en-tête de la méthode *identi-*

que à appeler était de la forme *Point identique (Point)*. Lors de l'exécution, la ligature dynamique tient compte du type de l'objet réellement référencé par *p1* (ici *Pointcol*) pour définir la classe à partir de laquelle se fera la recherche de la méthode voulue. Mais comme dans *Pointcol*, la méthode *identique* n'a pas la signature voulue, on poursuit la recherche dans les classes ascendantes et, finalement, on utilise la méthode *identifie* de *Point*. D'où le résultat constaté.

7 La super-classe Object

Jusqu'ici, nous pouvons considérer que nous avons défini deux sortes de classes : des classes simples et des classes dérivées.

En réalité, il existe une classe nommée *Object* dont dérive implicitement toute classe simple. Ainsi, lorsque vous définissez une classe *Point* de cette manière :

```
class Point
{ .....
}
```

tout se passe en fait comme si vous aviez écrit (vous pouvez d'ailleurs le faire) :

```
class Point extends Object
{ .....
}
```

Voyons les conséquences de cette propriété.

7.1 Utilisation d'une référence de type Object

Compte tenu des possibilités de compatibilité exposées précédemment, une variable de type *Object* peut être utilisée pour référencer un objet de type quelconque :

```
Point p = new Point (...) ;
Pointcol pc = new Pointcol (...) ;
Fleur f = new Fleur (...) ;
Object o ;
   .....
o = p ;    // OK
o = pc ;   // OK
o = f ;    // OK
```

Cette particularité peut être utilisée pour manipuler des objets dont on ne connaît pas le type exact (au moins à un certain moment). Cela pourrait être le cas d'une méthode qui se contente de transmettre à une autre méthode une référence qu'elle a reçue en argument.

Bien entendu, dès qu'on souhaitera appliquer une méthode précise à un objet référencé par une variable de type *Object*, il faudra obligatoirement effectuer une conversion appropriée.

Voyez cet exemple où l'on suppose que la classe *Point* dispose de la méthode *deplace* :

```
Point p = new Point (...) ;
Object o ;
   .....
o = p ;
o.deplace() ;            // erreur de compilation
((Point)o).deplace() ;   // OK en compilation (attention aux parenthèses)
Point p1 = (Point) o ;   // OK : idem ci-dessus, avec création d'une référence
p1.deplace() ;           //    intermédiaire dans p1
```

Notez bien les conséquences des règles relatives au polymorphisme. Pour pouvoir appeler une méthode *f* par *o.f()*, il ne suffit pas que l'objet effectivement référencé par *o* soit d'un type comportant une méthode *f*, il faut aussi que ladite méthode existe déjà dans la classe *Object*. Ce n'est manifestement pas le cas de la méthode *deplace*.

7.2 Utilisation de méthodes de la classe Object

La classe *Object* dispose de quelques méthodes qu'on peut soit utiliser telles quelles, soit redéfinir. Les plus importantes sont *toString* et *equals*.

7.2.1 La méthode toString

Elle fournit une chaîne, c'est-à-dire un objet de type *String*. Cette classe sera étudiée ultérieurement mais, comme on peut s'y attendre, un objet de type *String* contient une suite de caractères. La méthode *toString* de la classe *Object* fournit une chaîne contenant :

- le nom de la classe concernée,
- l'adresse de l'objet en hexadécimal (précédée de @).

Voyez ce petit programme :

```
class Point
{ public Point(int abs, int ord)
  { x = abs ; y = ord ;
  }
  private int x, y ;
}
public class ToString1
{ public static void main (String args[])
  { Point a = new Point (1, 2) ;
    Point b = new Point (5, 6) ;
    System.out.println ("a = " + a.toString()) ;
    System.out.println ("b = " + b.toString()) ;
  }
}
```

```
a = Point@fc17aedf
b = Point@fc1baedf
```

Exemple d'utilisation d'utilisation de la méthode toString

Il est intéressant de noter que le nom de classe est bien le nom de la classe correspondant à l'objet référencé, même si *toString* n'a pas été redéfinie. En effet, la méthode *toString* de la classe *Object* utilise une technique dite de "fonction de rappel", analogue à celle que nous avons employée avec la méthode *identifie* de l'exemple du paragraphe 6.3. Plus précisément, elle appelle une méthode *getClass*[1] qui fournit la classe de l'objet référencé sous forme d'un objet de type *Class* contenant, entre autres, le nom de la classe.

Bien entendu, vous pouvez toujours redéfinir la méthode *toString* à votre convenance. Par exemple, dans notre classe *Point*, nous pourrions lui faire fournir une chaîne contenant les coordonnées du point...

La méthode *toString* d'une classe possède en outre la particularité d'être automatiquement appelée en cas de besoin d'une conversion implicite en chaîne. Nous verrons que ce sera le cas de l'opérateur + lorsqu'il dispose d'un argument de type chaîne, ce qui vous permettra d'écrire par exemple :

```
System.out.println ("mon objet = " + o) ;
```

et ce, quels que soient le type de la référence *o* et celui de l'objet effectivement référencé par *o*.

7.2.2 La méthode equals

La méthode *equals* définie dans la classe *Object* se contente de comparer les adresses des deux objets concernés. Ainsi, avec :

```
Object o1 = new Point (1, 2) ;
Object o2 = new Point (1, 2) ;
```

L'expression *o1.equals(o2)* a pour valeur *false*.

On peut bien sûr redéfinir cette méthode à sa guise dans n'importe quelle classe. Toutefois, il faudra tenir compte des limitations du polymorphisme évoquées au paragraphe 6.9. Ainsi, avec :

```
class Point
{ .....
  boolean equals (Point p) { return ((p.x==x) && (p.y==y)) ; }
}
Point a = new Point (1, 2) ;
Point b = new Point (1, 2) ;
```

l'expression *a.equals(b)* aura bien sûr la valeur *true*. En revanche, avec :

1. Cette méthode est introduite automatiquement par Java dans toutes les classes.

```
Object o1 = new Point (1, 2) ;
Object o2 = new Point (1, 2) ;
```

l'expression *o1.equals(o2)* aura la valeur *false* car on aura utilisé la méthode *equals* de *Object* et non celle de *Point*.

8 Les membres protégés

Nous avons déjà vu qu'il existe différents droits d'accès aux membres d'une classe : public (mot clé *public*), privé (mot clé *private*), de paquetage (aucune mention).

Il existe un quatrième droit d'accès dit protégé (mot clé *protected*). Mais curieusement, les concepteurs de Java le font intervenir à deux niveaux totalement différents : le paquetage de la classe d'une part, ses classes dérivées d'autre part.

En effet, un membre déclaré *protected* est accessible à des classes du même paquetage, ainsi qu'à ses classes dérivées (qu'elles appartiennent ou non au même paquetage). Cette particularité complique quelque peu la conception des classes, ce qui fait qu'en pratique, ce droit d'accès est peu employé.

En C++

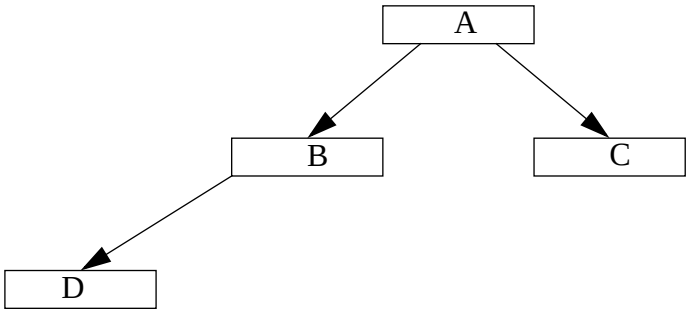
On peut déclarer des membres protégés (à l'aide du mot clé *protected*). Ils ne sont alors accessibles qu'aux classes dérivées, ce qui fait que l'accès protégé est beaucoup plus usité en C++ qu'en Java.

Informations complémentaires

Considérez une classe *A* ainsi définie :

```
class A
{ .....
  protected int n ;
}
```

et quelques descendantes de *A* sous la forme suivante :



Dans ces conditions :

– *B* accède à *n* de *A*,

– *D* accède à *n* de *B* ou de *A*,

– mais *C* n'accède pas à *n* de *B* (sauf si *B* et *C* sont dans le même paquetage) car aucun lien de dérivation ne relie *B* et *C*.

9 Cas particulier des tableaux

Jusqu'ici, nous avons considéré les tableaux comme des objets. Cependant, il n'est pas possible de définir exactement leur classe. En fait les tableaux ne jouissent que d'une partie des propriétés des objets.

1. Un tableau peut être considéré comme appartenant à une classe dérivée de *Object* :

```
Object o ;
o = new int [5] ;     // correct
   .....
o = new float [3] ;   // OK
```

2. Le polymorphisme peut s'appliquer à des tableaux d'objets. Plus précisément, si *B* dérive de *A*, un tableau de *B* est compatible avec un tableau de *A* :

```
class B extends A { ..... }
A ta[] ;
B tb[] ;
   .....
ta = tb ;     // OK car B dérive de A
tb = ta ;     // erreur
```

Malheureusement, cette propriété ne peut pas s'appliquer aux types primitifs :

```
int ti[] ; float tf[] ;
   .....
ti = tf ;   // erreur (on s'y attend car float n'est pas compatible avec int)
tf = ti ;   // erreur bien que int soit compatible avec float
```

3. Il n'est pas possible de dériver une classe d'une hypothétique classe tableau :

```
class Bizarre extends int []     // erreur
```

10 Classes et méthodes finales

Nous avons déjà vu comment le mot clé *final* pouvait s'appliquer à des variables locales ou à des champs d'une classe. Il interdit la modification de leur valeur. Ce mot clé peut aussi s'appliquer à une méthode ou à une classe, mais avec une signification totalement différente :

> Une méthode déclarée *final* ne peut pas être redéfinie dans une classe dérivée.

Le comportement d'une méthode finale[1] est donc complètement défini et il ne peut plus être remis en cause, sauf si la méthode appelle elle-même une méthode qui n'est pas déclarée *final*.

Une classe déclarée *final* ne peut plus être dérivée

On est ainsi certain que le contrat de la classe sera respecté.

On pourrait croire qu'une classe finale est équivalente à une classe non finale dont toutes les méthodes seraient finales. En fait, ce n'est pas vrai car :

- ne pouvant plus être dérivée, une classe finale ne pourra pas se voir ajouter de nouvelles fonctionnalités,
- une classe non finale dont toutes les méthodes sont finales pourra toujours être dérivée, donc se voir ajouter de nouvelles fonctionnalités.

Par son caractère parfaitement défini, une méthode finale permet au compilateur :

- de détecter des anomalies qui, sans cela, n'apparaîtraient que lors de l'exécution,
- d'optimiser certaines parties de code : appels plus rapides puisque indépendants de l'exécution, mise "en ligne" du code de certaines méthodes...

En revanche, il va de soi que le choix d'une méthode ou d'une classe finale est très contraignant et ne doit être fait qu'en toute connaissance de cause.

11 Les classes abstraites

11.1 Présentation

Une classe abstraite est une classe qui ne permet pas d'instancier des objets. Elle ne peut servir que de classe de base pour une dérivation. Elle se déclare ainsi :

```
abstract class A
{ .....
}
```

Dans une classe abstraite, on peut trouver classiquement des méthodes et des champs, dont héritera toute classe dérivée. Mais on peut aussi trouver des méthodes dites abstraites, c'est-à-dire dont on ne fournit que la signature et le type de la valeur de retour. Par exemple :

```
abstract class A
{ public void f() { ..... }         // f est définie dans A
  public abstract void g(int n) ;   // g n'est pas définie dans A ; on n'en
                                    // a fourni que l'en-tête
}
```

1. Nous commettrons l'abus de langage qui consiste à parler d'une classe finale pour désigner une classe déclarée avec l'attribut *final*.

Bien entendu, on pourra déclarer une variable de type *A* :

```
A a ;     // OK : a n'est qu'une référence sur un objet de type A ou dérivé
```

En revanche, toute instanciation d'un objet de type *A* sera rejetée par le compilateur :

```
a = new A(...) ; // erreur : pas d'instanciation d'objets d'une classe abstraite
```

En revanche, si on dérive de *A* une classe *B* qui définit la méthode abstraite *g* :

```
class B extends A
{ public void g(int n) { ..... }   // ici, on définit g
     .....
}
```

on pourra alors instancier un objet de type *B* par *new B(...)* et même affecter sa référence à une variable de type *A* :

```
A a = new B(...) ;       // OK
```

11.2 Quelques règles

1. Dès qu'une classe comporte une ou plusieurs méthodes abstraites, elle est abstraite, et ce même si l'on n'indique pas le mot clé *abstract* devant sa déclaration (ce qui reste quand même vivement conseillé). Ceci est correct :

   ```
   class A
   { public abstract void f() ;   // OK
     .....
   }
   ```

   Malgré tout, *A* est considérée comme abstraite et une expression telle que *new A(...)* sera rejetée.

2. Une méthode abstraite doit obligatoirement être déclarée *public*, ce qui est logique puisque sa vocation est d'être redéfinie dans une classe dérivée.

3. Dans l'en-tête d'une méthode déclarée abstraite, les noms d'arguments muets doivent figurer (bien qu'ils ne servent à rien) :

   ```
   abstract class A
   {  public abstract void g(int) ; // erreur : nom d'argument (fictif) obligatoire
   }
   ```

4. Une classe dérivée d'une classe abstraite n'est pas obligée de (re)définir toutes les méthodes abstraites de sa classe de base (elle peut même n'en redéfinir aucune). Dans ce cas, elle reste simplement abstraite (bien que ce ne soit pas obligatoire, il est conseillé de mentionner *abstract* dans sa déclaration) :

   ```
   abstract class A
   { public abstract void f1() ;
     public abstract void f2 (char c) ;
       .....
   }
   ```

```
abstract class B extends A      // abstract non obligatoire ici, mais conseillé
{ public void f1 () { ..... }   // définition de f1
    .....                       // pas de définition de f2
}
```

Ici, *B* définit *f1*, mais pas *f2*. La classe *B* reste abstraite (même si on ne l'a pas déclarée ainsi).

5. Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites. Notez que, toutes les classes dérivant de *Object*, nous avons utilisé implicitement cette règle dans tous les exemples précédents.

Remarque

L'utilisation du même mot clé *abstract* pour les classes et pour les méthodes fait que l'on parle en Java de classes abstraites et de méthodes abstraites. En Programmation Orientée Objet, on parle aussi de classe abstraite pour désigner une classe non instanciable ; en revanche, on parle généralement de méthode différée ou retardée pour désigner une méthode qui doit être redéfinie dans une classe dérivée.

11.3 Intérêt des classes abstraites

Le recours aux classes abstraites facilite largement la Conception Orientée Objet. En effet, on peut placer dans une classe abstraite toutes les fonctionnalités dont on souhaite disposer pour toutes ses descendantes :

- soit sous forme d'une implémentation complète de méthodes (non abstraites) et de champs (privés ou non) lorsqu'ils sont communs à toutes ses descendantes,
- soit sous forme d'interface de méthodes abstraites dont on est alors sûr qu'elles existeront dans toute classe dérivée instanciable.

C'est cette certitude de la présence de certaines méthodes qui permet d'exploiter le polymorphisme, et ce dès la conception de la classe abstraite, alors même qu'aucune classe dérivée n'a peut-être encore été créée. Notamment, on peut très bien écrire des canevas recourant à des méthodes abstraites. Par exemple, si vous avez défini :

```
abstract class X
{ public abstract void f() ;    // ici, f n'est pas encore définie
   .....
}
```

vous pourrez écrire une méthode (d'une classe quelconque) telle que :

```
void algo (X x)
{ .....
  x.f() ;    // appel correct ; accepté en compilation
  .....      // on est sûr que tout objet d'une classe dérivée de X
             // disposera bien d'une méthode f
}
```

Bien entendu, la redéfinition de *f* devra, comme d'habitude, respecter la sémantique prévue dans le contrat de *X*.

11.4 Exemple

Voici un exemple de programme illustrant l'emploi d'une classe abstraite nommée *Affichable*, dotée d'une seule méthode abstraite *affiche*. Deux classes *Entier* et *Flottant* dérivent de cette classe. La méthode *main* utilise un tableau hétérogène d'objets de type *Affichable* qu'elle remplit en instanciant des objets de type *Entier* et *Flottant*.

```
abstract class Affichable
{ abstract public void affiche() ;
}

class Entier extends Affichable
{ public Entier (int n)
  { valeur = n ;
  }
  public void affiche()
  { System.out.println ("Je suis un entier de valeur " + valeur) ;
  }
  private int valeur ;
}

class Flottant extends Affichable
{ public Flottant (float x)
  { valeur = x ;
  }
  public void affiche()
  { System.out.println ("Je suis un flottant de valeur " + valeur) ;
  }
  private float valeur ;
}

public class Tabhet3
{ public static void main (String[] args)
  { Affichable [] tab ;
    tab = new Affichable [3] ;
    tab [0] = new Entier (25) ;
    tab [1] = new Flottant (1.25f) ; ;
    tab [2] = new Entier (42) ;
    int i ;
    for (i=0 ; i<3 ; i++)
      tab[i].affiche() ;
  }
}
```

```
Je suis un entier de valeur 25
Je suis un flottant de valeur 1.25
Je suis un entier de valeur 42
```

*Exemple d'utilisation d'une classe abstraite (*Affichable*)*

Remarque

Un classe abstraite peut ne comporter que des méthodes abstraites et aucun champ. C'est d'ailleurs ce qui se produit ici. Dans ce cas, nous verrons qu'une interface peut jouer le même rôle ; nous montrerons comment transformer dans ce sens l'exemple précédent.

12 Les interfaces

Nous venons de voir comment une classe abstraite permettait de définir dans une classe de base des fonctionnalités communes à toutes ses descendantes, tout en leur imposant de redéfinir certaines méthodes. Si l'on considère une classe abstraite n'implantant aucune méthode et aucun champ (hormis des constantes), on aboutit à la notion d'interface. En effet, une interface définit les en-têtes d'un certain nombre de méthodes, ainsi que des constantes. Cependant, nous allons voir que cette dernière notion se révèle plus riche qu'un simple cas particulier de classe abstraite. En effet :

- une classe pourra implémenter plusieurs interfaces (alors qu'une classe ne pouvait dériver que d'une seule classe abstraite),
- la notion d'interface va se superposer à celle de dérivation, et non s'y substituer,
- les interfaces pourront se dériver,
- on pourra utiliser des variables de type interface.

Commençons par voir comment on définit une interface et comment on l'utilise, avant d'en étudier les propriétés en détail.

12.1 Mise en œuvre d'une interface

12.1.1 Définition d'une interface

La définition d'une interface se présente comme celle d'une classe. On y utilise simplement le mot clé *interface* à la place de *class* :

```
public interface I
{ void f(int n) ;   // en-tête d'une méthode f (public abstract facultatifs)
  void g() ;        // en-tête d'une méthode g (public abstract facultatifs)
}
```

Une interface peut être dotée des mêmes droits d'accès qu'une classe (*public* ou droit de paquetage).

Dans la définition d'une interface, on ne peut trouver que des en-têtes de méthodes (cas de *f* et *g* ici) ou des constantes (nous reviendrons plus loin sur ce point). Par essence, les méthodes d'une interface sont abstraites (puisqu'on n'en fournit pas de définition) et publiques (puisqu'elles devront être redéfinies plus tard). Néanmoins, il n'est pas nécessaire de mentionner les mots clés *public* et *abstract* (on peut quand même le faire).

12.1.2 Implémentation d'une interface

Lorsqu'on définit une classe, on peut préciser qu'elle implémente une interface donnée en utilisant le mot clé *implements*, comme dans :

```
class A implements I
{ // A doit (re)définir les méthodes f et g prévues dans l'interface I
}
```

Ici, on indique que *A* doit définir les méthodes prévues dans l'interface *I*, c'est-à-dire *f* et *g*. Si cela n'est pas le cas, on obtiendra une erreur de compilation (attention : on ne peut pas différer cette définition de méthode, comme on pourrait éventuellement le faire dans le cas d'une classe abstraite).

Une même classe peut implémenter plusieurs interfaces :

```
public interface I1
{ void f() ;
}
public interface I2
{ int h() ;
}
class A implements I1, I2
{ // A doit obligatoirement définir les méthodes f et h prévues dans I1 et I2
}
```

12.2 Variables de type interface et polymorphisme

Bien que la vocation d'une interface soit d'être implémentée par une classe, on peut définir des variables de type interface :

```
public interface I { .....}
   .....
I i ; // i est une référence à un objet d'une classe implémentant l'interface I
```

Bien entendu, on ne pourra pas affecter à *i* une référence à quelque chose de type *I* puisqu'on ne peut pas instancier une interface (pas plus qu'on ne pouvait instancier une classe abstraite !). En revanche, on pourra affecter à *i* n'importe quelle référence à un objet d'une classe implémentant l'interface *I* :

```
class A implements I { ..... }
   .....
I i = new A(...) ;   // OK
```

De plus, à travers *i*, on pourra manipuler des objets de classes quelconques, non nécessairement liées par héritage, pour peu que ces classes implémentent l'interface *I*.

Voici un exemple illustrant cet aspect. Une interface *Affichable* comporte une méthode *affiche*. Deux classes *Entier* et *Flottant* implémentent cette interface (aucun lien d'héritage n'apparaît ici). On crée un tableau hétérogène de références de "type" *Affichable* qu'on remplit en instanciant des objets de type *Entier* et *Flottant*. En fait, il s'agit d'une transposition de l'exemple de programme du paragraphe 11.4 qui utilisait des classes abstraites.

```
interface Affichable
{ void affiche() ;
}
class Entier implements Affichable
{ public Entier (int n)
  { valeur = n ;
  }
  public void affiche()
  { System.out.println ("Je suis un entier de valeur " + valeur) ;
  }
  private int valeur ;
}
class Flottant implements Affichable
{ public Flottant (float x)
  { valeur = x ;
  }
  public void affiche()
  { System.out.println ("Je suis un flottant de valeur " + valeur) ;
  }
  private float valeur ;
}
public class Tabhet4
{ public static void main (String[] args)
  { Affichable [] tab ;
    tab = new Affichable [3] ;
    tab [0] = new Entier (25) ;
    tab [1] = new Flottant (1.25f) ; ;
    tab [2] = new Entier (42) ;
    int i ;
    for (i=0 ; i<3 ; i++)
      tab[i].affiche() ;
  }
}
```

```
Je suis un entier de valeur 25
Je suis un flottant de valeur 1.25
Je suis un entier de valeur 42
```

Exemple d'utilisation de variables de type interface

Cet exemple est restrictif puisqu'il peut se traiter avec une classe abstraite. Voyons maintenant ce que l'interface apporte de plus.

Remarque

Ici, notre interface a été déclarée avec un droit de paquetage, et non avec l'attribut *public*, ce qui nous a permis de placer sa définition dans le même fichier source que les autres classes (les droits d'accès des interfaces sont en effet régis par les mêmes règles que ceux des classes). En pratique, il en ira rarement ainsi, dans la mesure où chaque interface disposera de son propre fichier source.

12.3 Interface et classe dérivée

La clause *implements* est une garantie qu'offre une classe d'implémenter les fonctionnalités proposées dans une interface. Elle est totalement indépendante de l'héritage ; autrement dit, une classe dérivée peut implémenter une interface (ou plusieurs) :

```
interface I
{ void f(int n) ;
  void g() ;
}
class A { ..... }
class B extends A implements I
{    // les mméthodes f et g doivent soit être déjà définies dans A,
     // soit définies dans B
}
```

On peut même rencontrer cette situation :

```
interface I1 { ..... }
interface I2 { ..... }
class A implements I1 { ..... }
class B extends A implements I2 { ..... }
```

12.4 Interfaces et constantes

L'essentiel du concept d'interface réside dans les en-têtes de méthodes qui y figurent. Mais une interface peut aussi renfermer des constantes symboliques qui seront alors accessibles à toutes les classes implémentant l'interface :

```
interface I
{ void f(int n) ;
  void g() ;
  static final int MAXI = 100 ;
}
class A implements I
{ // doit définir f et g
  // dans toutes les méthodes de A, on a accès au symbole MAXI :
  // par exemple :    if (i < MAXI) .....
}
```

Ces constantes sont automatiquement considérées comme si elles avaient été déclarées *static* et *final*. Il doit s'agir obligatoirement d'expressions constantes.

Elles sont accessibles en dehors d'une classe implémentant l'interface. Par exemple, la constante *MAXI* de l'interface *I* se notera simplement *I.MAXI*.

Remarque

On peut dire que, dans une interface, les méthodes et les constantes sont considérées de manière opposée. En effet, les méthodes doivent être implémentées par la classe, tandis que les constantes sont utilisables par la classe.

12.5 Dérivation d'une interface

On peut définir une interface comme une généralisation d'une autre. On utilise là encore le mot clé *extends*, ce qui conduit à parler d'héritage ou de dérivation, et ce bien qu'il ne s'agisse en fait que d'emboîter simplement des déclarations :

```
interface I1
{ void f(int n) ;
  static final int MAXI = 100 ;
}
interface I2 extends I1
{ void g() ;
  static final int MINI = 20 ;
}
```

En fait, la définition de *I2* est totalement équivalente à :

```
interface I2
{ void f(int n) ;
  void g() ;
  static final int MAXI = 100 ;
  static final int MINI = 20 ;
}
```

La dérivation des interfaces revient simplement à concaténer des déclarations. Il n'en va pas aussi simplement pour l'héritage de classes, où les notions d'accès deviennent fondamentales.

12.6 Conflits de nom

Considérons :

```
interface I1
{ void f(int n) ;
  void g() ;
}
interface I2
{ void f(float x) ;
  void g() ;
}
```

```
class A implements I1, I2
{ // A doit définir deux méthodes f : void f(int) et void f(float),
  // mais une seule méthode g : void g()
```

En ce qui concerne la méthode *f*, on voit que, pour implémenter *I1* et *I2*, la classe *A* doit simplement la surdéfinir convenablement. En ce qui concerne la méthode *g*, en revanche, il semble qu'un conflit de noms apparaisse. En fait, il n'en est rien puisque les deux interfaces *I1* et *I2* définissent le même en-tête de méthode *g* ; il suffit donc que *A* définisse la méthode requise (même si elle elle demandée deux fois !).

En revanche, considérons maintenant :

```
interface I1
{ void f(int n) ;
  void g() ;
}
interface I2
{ void f(float x) ;
  int g() ;
}
class A implements I1, I2
{ // pour satisfaire à I1 et I2, A devrait contenir à la fois une méthode
  // void g() et une méthode int g(), ce qui n'est pas possible
  // d'après les règles de redéfinition
}
```

Cette fois, une classe ne peut implémenter à la fois *I1* et *I2*.

Informations complémentaires

Au Chapitre 10, nous verrons qu'une méthode peut spécifier par *throws* les exceptions qu'elles est susceptible de déclencher. Rien n'empêche que les clauses *throws* d'une méthode donnée diffèrent d'une interface à une autre ; par exemple, on pourrait avoir :

```
void g() throws E1, E2 ;
```

dans *I1* et :

```
void g() throws E1, E3 ;
```

dans *I2*.

Dans ce cas, l'implémentation de *g* dans *A* ne devra pas spécifier plus d'exceptions que n'en spécifie chacune des interfaces. Ainsi, on pourra rencontrer l'une de ces possibilités :

```
void g() throws E1 { ..... }
void g() { ..... }
```

12.7 L'interface Cloneable

Java dispose de quelques outils destinés à faciliter la gestion du clonage des objets (copie profonde).

Tout d'abord, la classe *Object* possède une méthode *clone* protégée qui se contente d'effectuer une copie superficielle de l'objet. L'idée des concepteurs de Java est que cette méthode doit être redéfinie dans toute classe *clonable*

Par ailleurs, il existe une interface très particulière *Cloneable*. Ce nom s'emploie comme un nom d'interface dans une clause *implements*. Mais, contrairement à ce qui se produirait avec une interface usuelle, cette clause n'impose pas la redéfinition de la méthode *clone*.

En fait, la déclaration :

```
class X implements Cloneable
```

mentionne que la classe *X* peut subir une copie profonde par appel de la méthode *clone*. Cette dernière peut être celle de *Object* ou une méthode fournie par *X*.

Enfin, une tentative d'appel de *clone* sur une classe n'implémentant pas l'interface *Cloneable* conduit à une exception *CloneNotSupportedException*. Vous pouvez également lever vous-même une telle exception si vous décidez qu'un objet d'une classe (implémentant l'interface *Cloneable*) n'est pas copiable ; vous réalisez ainsi du *clonage conditionnel*.

Notez que l'en-tête de *clone* est :

```
Object clone() ;
```

Cela signifie que son utilisation nécessite toujours un *cast* de son résultat dans le type effectif de l'objet soumis à copie.

13 Les classes enveloppes

Il nous est déjà arrivé de recourir à des notations telles que *Integer.MAX_VALUE* ou *Double.MIN_VALUE*.

Il existe en effet des classes nommées *Boolean*, *Byte*, *Character*, *Short*, *Integer*, *Long*, *Float* et *Double*, destinées à manipuler des valeurs d'un type primitif en les encapsulant dans une classe. Cela permet de disposer de méthodes et de compenser le fait que les types primitifs ne soient pas des classes.

Toutes ces classes disposent d'un constructeur recevant un argument d'un type primitif :

```
Integer ObjInt = new Integer (5) ;   // ObjInt contient la référence à un
                          // objet de type Integer encapsulant la valeur 5
```

Elles disposent toutes d'une méthode de la forme *xxxValue* (*xxx* représentant le nom du type primitif) qui permet de retrouver la valeur dans le type primitif correspondant :

```
Integer nObj = new Integer (12) ;
Double xObj= new Double (5.25) ;
   .....
int n = nObj.intValue() ;          // n contient 12
double x = xObj.doubleValue() ;    // x contient 5.25
```

Nous verrons aussi dans le chapitre consacré aux chaînes qu'elles disposent d'une méthode *toString* effectuant la conversion de la valeur qu'elles contiennent en une chaîne, ainsi que d'une méthode de la forme *parseXXX* permettant de convertir une chaîne en un type primitif.

14 Eléments de conception des classes

Voici quelques réflexions concernant l'héritage et les interfaces, qui peuvent intervenir dans la conception de vos classes.

14.1 Respect du contrat

Nous avons déjà dit qu'une classe constituait une sorte de contrat défini par les interfaces des méthodes publiques (signatures et valeur de retour) et leur sémantique (leur rôle). Grâce à l'encapsulation des données, l'utilisateur d'une classe n'a pas à en connaître l'implémentation (champs de données ou corps des méthodes).

En principe, ce contrat doit être respecté en cas de dérivation. Lorsqu'on surdéfinit une méthode, on s'arrange pour en conserver la sémantique. On notera bien qu'à ce niveau, aucun outil ne permet actuellement de s'assurer que ce principe est respecté.

En général, on a toujours intérêt à doter une classe dérivée d'un constructeur. Même si ce dernier n'a rien de particulier à faire, on se contentera d'un appel d'un constructeur de la classe de base.

14.2 Relations entre classes

Nous avons déjà fait remarquer que l'héritage créait une relation de type "est". Si *T'* dérive de *T*, un objet de type *T'* peut aussi être considéré comme un objet de type *T*. Cette propriété est à la base du polymorphisme.

Nous avons aussi rencontré un autre type de relation entre objets, à savoir la relation de type "a" (appartenance ou composition) correspondant à la situation d'objets membres. Dans ce cas, si la classe *T* comporte un champ de type *U*, un objet de type *T* possède un champ qui est un objet de type *U*.

D'autres relations peuvent apparaître entre les classes, notamment celle d'utilisation. Mais les deux précédentes sont les plus importantes. Dans certaines circonstances, il est nécessaire d'opérer avec soin un choix entre elles. Pour illustrer cela, supposez que nous disposions d'une classe *Point* usuelle :

```
class Point
{ .....
  public void deplace (...) { ..... }
  private int x, y ;
}
```

Pour définir une classe *Pointcol*, nous pouvons procéder comme nous l'avons fait jusqu'ici en mettant en oeuvre une relation "est" :

```
class Pointcol extends Point    // Pointcol "est" un Point
{ .....
  private byte couleur ;
}
```

Dans ce cas, avec :

```
Pointcol pc ;
```

comme nous l'avons vu, un objet de type *Pointcol* est un objet de type *Point* et nous pouvons lui appliquer les méthodes publiques de *Point* :

```
pc.deplace (...) ;
```

Mais nous aurions aussi pu considérer qu'un point coloré est formé d'un point et d'une couleur et définir notre classe *Pointcol* de cette façon :

```
class Pointcol
{ .....
  private Point p ;      // relation "a"
  private byte couleur ;
}
```

Dans ce cas, toujours avec :

```
Pointcol pc ;
```

on voit qu'il n'est plus possible de déplacer le point. Il faudrait pour cela pouvoir procéder ainsi :

```
pc.p.deplace (...) ;  // impossible : le champ p est privé
```

Même si l'accès à *p* était possible (par exemple si *p* était protégé ou public), la démarche à employer serait différente de la précédente. En effet, il faudrait déplacer le membre *p* du point coloré de référence *pc* et non plus directement le point coloré de référence *pc*.

Comme on s'en doute, les différences entre les deux types de relation seront encore plus criantes si l'on considère les possibilités de polymorphisme. Ces dernières ne seront exploitables que dans le premier cas (relation "est").

D'une manière générale, le choix entre les deux types de relation n'est pas toujours aussi facile que dans cet exemple.

14.3 Différences entre interface et héritage

Lorsqu'elle n'est pas abstraite, la classe de base fournit des implémentations complètes de méthodes.

Une interface fournit simplement un contrat à respecter sous forme d'en-têtes de méthodes. La classe implémentant l'interface est responsable de leur implémentation. Des classes différentes peuvent implémenter différemment une même interface, alors que des classes dérivées d'une même classe de base en partagent la même implémentation.

On dit souvent que Java ne dispose pas de l'héritage multiple mais que ce dernier peut être avantageusement remplacé par l'utilisation d'interfaces. On voit maintenant que cette affirmation doit être nuancée. En effet, une classe implémentant plusieurs interfaces doit fournir du code (et le tester !) pour l'implémentation des méthodes correspondantes. Les interfaces multiples assurent donc une aide manifeste à la conception en assurant le respect du contrat. En revanche, elles ne simplifient pas le développement du code, comme le permet l'héritage multiple.

15 Les classes anonymes

Java permet de définir ponctuellement une classe, sans lui donner de nom. Cette particularité a été introduite par la version 1.1 pour faciliter la gestion des événements. Nous la présenterons succinctement ici, en dehors de ce contexte.

15.1 Exemple de classe anonyme

Supposons que l'on dispose d'une classe *A*. Il est possible de créer un objet d'une classe dérivée de *A*, en utilisant une syntaxe de cette forme :

```
A a ;
a = new A() {  // champs et méthodes qu'on introduit dans
               // la classe anonyme dérivée de A
            } ;
```

Tout se passe comme si l'on avait procédé ainsi :

```
A a ;
class A1 extends A  {  // champs et méthodes spécifiques à A1
                    } ;
   .....
a = new A1() ;
```

Cependant, dans ce dernier cas, il serait possible de définir des références de type *A1*, ce qui n'est pas possible dans le premier cas.

Voici un petit programme illustrant cette possibilité. La classe *A* y est réduite à une seule méthode *affiche*. Nous créons une classe anonyme, dérivée de *A*, qui redéfinit la méthode *affiche*.

```
class A
{ public void affiche()  { System.out.println ("Je suis un A") ;
                         }
}
public class Anonym1
{ public static void main (String[] args)
  { A a ;
    a = new A() { public void affiche ()
                  { System.out.println ("Je suis un anonyme derive de A") ;
                  }
                } ;
    a.affiche() ;
  }
}
```

```
Je suis un anonyme derive de A
```

Exemple d'utilisation d'une classe anonyme dérivée d'une autre

Notez bien que si *A* n'avait pas comporté de méthode *affiche*, l'appel *a.affiche()* aurait été incorrect, compte tenu du type de la référence *a* (revoyez éventuellement les règles relatives au polymorphisme). Cela montre qu'une classe anonyme ne peut pas introduire de nouvelles méthodes ; notez à ce propos que même un appel de cette forme serait incorrect :

```
( new A() { public void affiche ()
                  { System.out.println ("Je suis un anonyme derive de A") ;
                  }
          }).affiche() ;
```

15.2 Les classes anonymes d'une manière générale

15.2.1 Il s'agit de classes dérivées ou implémentant une interface

La syntaxe de définition d'une classe anonyme ne s'applique que dans deux cas :

- classe anonyme dérivée d'une autre (comme dans l'exemple précédent),
- classe anonyme implémentant une interface.

Voici un exemple simple de la deuxième situation :

```
interface Affichable
{ public void affiche() ;
}
public class Anonym2
{ public static void main (String[] args)
  { Affichable a ;
    a = new Affichable()
          { public void affiche ()
            { System.out.println ("Je suis un anonyme implementant Affichable") ;
           }
          } ;
    a.affiche() ;
  }
}
```

```
Je suis un anonyme implementant Affichable
```

Exemple d'utilisation d'une classe anonyme implémentant une interface

15.2.2 Utilisation de la référence à une classe anonyme

Dans les précédents exemples, la référence de la classe anonyme était conservée dans une variable (d'un type de base ou d'un type interface). On peut aussi la transmettre en argument d'une méthode ou en valeur de retour.Voyez cet exemple :

```
interface I
{ ..... }
public class Util
{ public static f(I i) { ..... }
}
   .....
f(new I() { // implémentation des méthodes de I } ) ;
```

L'utilisation des classes anonymes conduit généralement à des codes peu lisibles. On la réservera à des cas très particuliers où la définition de la classe anonyme reste brève.

Nous en rencontrerons quelques exemples dans la définition de classes écouteurs d'événements.