Claude Delannoy

Programmer en Java

6e édition

© Groupe Eyrolles, 2000-2009,
ISBN : 978-2-212-12623-5
EYROLLES

6

Les classes et les objets

Le premier chapitre a exposé de façon théorique les concepts de base de la P.O.O., en particulier ceux de classe et d'objet. Nous avons vu que la notion de classe généralise celle de type : une classe comporte à la fois des champs (ou données) et des méthodes. Quant à elle, la notion d'objet généralise celle de variable : un type classe donné permet de créer (on dit aussi *instancier*) un ou plusieurs objets du type, chaque objet comportant son propre jeu de données. En P.O.O pure, on réalise ce qu'on nomme l'encapsulation des données ; cela signifie qu'on ne peut pas accéder aux champs d'un objet autrement qu'en recourant aux méthodes prévues à cet effet.

Dans les précédents chapitres, nous avons vu comment mettre en œuvre une classe en Java. Mais toutes les classes que nous avons réalisées étaient très particulières puisque :

- nous ne créions aucun objet du type de la classe,
- nos classes ne comportaient qu'une seule méthode nommée *main*, et il ne s'agissait même pas d'une méthode au sens usuel car on pouvait exécuter ses instructions sans qu'on ait à préciser à quel objet elles devaient s'appliquer (cela en raison de la présence du mot-clé *static*). Nous vous avions d'ailleurs fait remarquer que cette méthode *main* était en fait semblable au programme principal ou à la fonction principale des autres langages.

Ici, nous allons aborder la notion de classe dans toute sa généralité, telle qu'elle apparaît dans les concepts de P.O.O.

Nous verrons tout d'abord comment définir une classe et l'utiliser en instanciant des objets du type correspondant, ce qui nous amènera à introduire la notion de référence à un objet. Nous étudierons ensuite l'importante notion de constructeur, méthode appelée automatiquement lors de la création d'un objet. Puis nous examinerons comment se présente l'affectation

d'objets et en quoi elle diffère de celle des variables d'un type primitif. Nous préciserons les propriétés des méthodes (arguments, variables locales..) avant d'aborder les possibilités de surdéfinition. Nous présenterons alors le mode de transmission des arguments ou des valeurs de retour ; nous verrons plus précisément que les valeurs d'un type de base sont transmises par valeur, tandis que les valeurs de type objet le sont par référence.

Nous étudierons ensuite ce que l'on nomme les champs et les méthodes de classe et nous verrons que la méthode *main* est de cette nature.

Après un exemple de classe possédant des champs eux-mêmes de type classe (objets membres), nous vous présenterons la notion de classe interne (introduite seulement par Java 1.1). Nous terminerons sur la notion de paquetage.

1 La notion de classe

Nous allons commencer par vous exposer les notions de classe, d'objet et d'encapsulation à partir d'un exemple simple de classe. Par souci de clarté, celle-ci ne comportera pas de constructeur ; en pratique, la plupart des classes en disposent. Cette notion de constructeur sera exposée séparément par la suite.

Nous verrons d'abord comment créer une classe, c'est-à-dire écrire les instructions permettant d'en définir le contenu (champs ou données) et le comportement (méthodes). Puis nous verrons comment utiliser effectivement cette classe au sein d'un programme.

1.1 Définition d'une classe Point

Nous vous proposons de définir une classe nommée *Point*, destinée à manipuler les points d'un plan.

Rappelons le canevas général de définition d'une classe en Java, que nous avons déjà utilisé dans les précédents chapitres dans le seul but de contenir une fonction *main* :

```
public class Point
{ // instructions de définition des champs et des méthodes de la classe
}
```

Nous reviendrons un peu plus loin sur le rôle exact de *public*. Pour l'instant, sachez simplement qu'il intervient dans l'accès d'autres classes à la classe *Point*. En son absence, l'accès à *Point* serait limité aux seules classes du même paquetage[1].

Voyons maintenant comment définir le contenu de notre classe, en distinguant les champs des méthodes.

1. Ce qui ne serait pas une limitation gênante si vous vous en tenez à l'utilisation du paquetage par défaut (autrement dit, si vous ne faites pas appel à l'instruction *package*).

1.1.1 Définition des champs

Nous supposerons ici qu'un objet de type *Point* sera représenté par deux coordonnées entières. Ils nous suffira de les déclarer ainsi :

```
private int x ;    // abscisse
private int y ;    // ordonnee
```

Notez la présence du mot-clé *private* qui précise que ces champs *x* et *y* ne seront pas accessibles à l'extérieur de la classe, c'est-à-dire en dehors de ses propres méthodes. Cela correspond à l'encapsulation des données, dont on voit qu'elle n'est pas obligatoire en Java (elle est toutefois fortement recommandée).

Ces déclarations peuvent être placées où vous voulez à l'intérieur de la définition de la classe, et pas nécessairement avant les méthodes. En général, on place les champs et méthodes privées à la fin.

1.1.2 Définition des méthodes

Supposons que nous souhaitions disposer des trois méthodes suivantes :

- *initialise* pour attribuer des valeurs aux coordonnées d'un point,
- *deplace* pour modifier les coordonnées d'un point,
- *affiche* pour afficher un point ; par souci de simplicité, nous nous contenterons ici d'afficher les coordonnées du point.

La définition d'une méthode ressemble à celle d'une procédure ou d'une fonction dans les autres langages, ou encore à la définition de la méthode *main* déjà rencontrée. Elle se compose d'un en-tête et d'un bloc. Ainsi, la définition de la méthode *initialise* pourra se présenter comme ceci :

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

L'en-tête précise :

- le *nom* de la méthode, ici *initialise* ;
- le *mode d'accès* : nous avons choisi *public* pour que cette méthode soit effectivement utilisable depuis un programme quelconque ; nous avons déjà rencontré *private* pour des champs ; nous aurons l'occasion de revenir en détails sur ces problèmes d'accès ;
- les *arguments* qui seront fournis à la méthode lors de son appel, que nous avons choisi de nommer *abs* et *ord* ; il s'agit d'arguments muets, identiques aux arguments muets d'une fonction ou d'une procédure d'un autre langage ;
- le *type de la valeur de retour* ; nous verrons plus tard qu'une méthode peut fournir un résultat, c'est-à-dire se comporter comme ce que l'on nomme une fonction dans la plupart des langages (et aussi en mathématiques) ; ici, notre méthode ne fournit aucun résultat, ce que l'on doit préciser en utilisant le mot-clé *void*.

Si nous examinons maintenant le corps de notre méthode *initialise*, nous y rencontrons une première affectation :

x = abs ;

Le symbole *abs* désigne (classiquement) la valeur reçue en premier argument. Quant à *x*, il ne s'agit ni d'un argument, ni d'une variable locale au bloc constituant la méthode. En fait, *x* désigne le champ *x* de l'objet de type *Point* qui sera effectivement concerné par l'appel de *initialise*. Nous verrons plus loin comment se fait cette association entre un objet donné et une méthode.

La seconde affectation de *initialise* est comparable à la première.

Les définitions des autres méthodes de la classe *Point* ne présentent pas de difficulté particulière. Voici la définition complète de notre classe *Point* :

```
public 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 un point de coordonnees " + x + " " + y) ;
  }
  private int x ;   // abscisse
  private int y ;   // ordonnee
}
```

Définition d'une classe Point

Remarque

Ici, tous les champs de notre classe *Point* étaient privés et toutes ses méthodes étaient publiques. On peut disposer de méthodes privées ; dans ce cas, elles ne sont utilisables que par d'autres méthodes de la classe. On peut théoriquement disposer de champs publics mais c'est fortement déconseillé. Par ailleurs, nous verrons qu'il existe également un mode d'accès dit "de paquetage", ainsi qu'un accès protégé (*protected*) partiellement lié à l'héritage.

En C++

En C++, on distingue la déclaration d'une classe de sa définition. Généralement, la première figure dans un fichier en-tête, la seconde dans un fichier source. Cette distinction n'existe pas en Java.

1.2 Utilisation de la classe Point

Comme on peut s'y attendre, la classe *Point* va permettre d'instancier des objets de type *Point* et de leur appliquer à volonté les méthodes publiques *initialise*, *deplace* et *affiche*.

Bien entendu, cette utilisation ne pourra se faire que depuis une autre méthode puisque, en Java, toute instruction appartient toujours à une méthode. Mais il pourra s'agir de la méthode particulière *main*, et c'est ainsi que nous procéderons dans notre exemple de programme complet.

1.2.1 La démarche

À l'intérieur d'une méthode quelconque, une déclaration telle que :

```
Point a ;
```

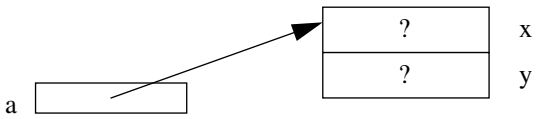
est tout à fait correcte. Cependant, contrairement à la déclaration d'une variable d'un type primitif (comme *int n ;*), elle ne réserve pas d'emplacement pour un objet de type *Point*, mais seulement un emplacement pour une *référence* à un objet de type *Point*. L'emplacement pour l'objet proprement dit sera alloué sur une demande explicite du programme, en faisant appel à un opérateur unaire nommé *new*. Ainsi, l'expression :

```
new Point()      // attention à la présence des parenthèses ()
```

crée un emplacement pour un objet de type *Point* et fournit sa référence en résultat. Par exemple, on pourra procéder à cette affectation :

```
a = new Point() ;   // crée d'un objet de type Point et place sa référence dans a
```

La situation peut être schématisée ainsi :



Pour l'instant, les champs *x* et *y* n'ont apparemment pas encore reçu de valeur (on verra plus tard qu'en réalité, ils ont été initialisés par défaut à 0).

Une fois qu'une référence à un objet a été convenablement initialisée, on peut appliquer n'importe quelle méthode à l'objet correspondant. Par exemple, on pourra appliquer la méthode *initialise* à l'objet référencé par *a*, en procédant ainsi :

```
a.initialise (3, 5) ;   // appelle la méthode initialise du type Point
                        // en l'appliquant à l'objet de référence a, et
                        // en lui transmettant les arguments 3 et 5
```

Si on fait abstraction du préfixe *a*, cet appel est analogue à un appel classique de fonction tel qu'on le rencontre dans la plupart des langages. Bien entendu, c'est ce préfixe qu va préciser à la méthode sur quel objet elle doit effectivement opérer. Ainsi, l'instruction *x = abs* de la méthode *initialise* placera la valeur reçue pour *abs* (ici 3) dans le champ *x* de l'objet *a*.

Remarque

Nous dirons que *a* est une variable de type classe. Nous ferons souvent l'abus de langage consistant à appeler objet *a* l'objet dont la référence est contenue dans *a*.

En C++

En C++, la déclaration d'un objet entraîne toujours la réservation d'un emplacement approprié (comme pour un type de base), contrairement à Java qui réserve un emplacement pour un type primitif, mais seulement une référence pour un objet. En revanche, en C++, on peut instancier un objet de deux manières différentes : par sa déclaration (objet automatique) ou par l'opérateur *new* (objet dynamique). Dans ce dernier cas, on obtient en résultat une adresse qu'on manipule par le biais d'un pointeur, ce dernier jouant un peu le rôle de la référence de Java. Le fait que Java ne dispose que d'un seul mode d'instanciation (correspondant aux objets dynamiques de C++) contribue largement à la clarté des programmes.

1.2.2 Exemple

Comme nous l'avons déjà dit, nous pouvons employer notre classe *Point* depuis toute méthode d'une autre classe, ou depuis une méthode *main*. Cette dernière doit de toute façon être elle aussi une méthode (statique) d'une classe. A priori, nous pourrions faire de *main* une méthode de notre classe *Point*. Mais la démarche serait alors trop particulière : nous préférons donc qu'elle appartienne à une autre classe. Voici un exemple complet d'une classe nommée *TstPoint* contenant (seulement) une fonction *main* utilisant notre classe *Point* :

```
public class TstPoint
{ public static void main (String args[])
  { Point a ;
    a = new Point() ;
    a.initialise(3, 5) ;   a.affiche() ;
    a.deplace(2, 0) ;      a.affiche() ;
    Point b = new Point() ;
    b.initialise (6, 8) ;  b.affiche() ;
  }
}
```

```
Je suis un point de coordonnees 3 5
Je suis un point de coordonnees 5 5
Je suis un point de coordonnees 6 8
```

Exemple d'utilisation de la classe Point

Notez que nous vous fournissons un exemple d'exécution, bien que pour l'instant nous ne vous ayons pas encore précisé comment exécuter un programme formé de plusieurs classes (ici *TstPoint* et *Point*), ce que nous ferons au paragraphe suivant.

Remarque

Dans notre classe *Point*, les champs *x* et *y* ont été déclarés privés. Une tentative d'utilisation directe, en dehors des méthodes de *Point*, conduirait à une erreur de compilation. Ce serait notamment le cas si, dans notre méthode *main*, nous cherchions à introduire des instructions telles que :

```
a.x = 5 ;                                    // erreur : x est privé
System.out.println ("ordonnee de a " + a.y) ;   // erreur : y est privé
```

1.3 Mise en œuvre d'un programme comportant plusieurs classes

Jusqu'ici, nos programmes étaient formés d'une seule classe. Il suffisait de la compiler et de lancer l'exécution. Avec plusieurs classes, les choses sont légèrement différentes et plusieurs démarches sont possibles. Nous commencerons par examiner la plus courante, à savoir utiliser un fichier source par classe.

1.3.1 Un fichier source par classe

Vous aurez sauvegardé le source de la classe *Point* dans un fichier nommé *Point.java*. Sa compilation donnera naissance au fichier de *byte codes Point.class*. Bien entendu, il n'est pas question d'exécuter directement ce fichier puisque la machine virtuelle recherche une fonction *main*.

En ce qui concerne la classe *TstPoint*, vous aurez là aussi sauvegardé son source dans un fichier *TstPoint.java*. Pour le compiler, il faut :

- que *Point.class* existe (ce qui est le cas si *Point.java* a été compilé sans erreurs),
- que le compilateur ait accès à ce fichier ; selon l'environnement utilisé, des problèmes de localisation du fichier peuvent se manifester ; la notion de paquetage peut aussi intervenir mais si, comme nous vous l'avons déjà conseillé, vous n'y avez pas fait appel, aucun problème ne se posera[1].

Une fois compilé *TstPoint*, il ne restera plus qu'à exécuter le fichier *TstPoint.class* ainsi obtenu.

1. Nous attirons à nouveau votre attention sur les "générateurs automatiques" de certains environnements qui introduisent d'office une instruction *package xxx* et qu'il est préférable de supprimer.

Bien entendu, la démarche à utiliser pour procéder à ces différentes étapes dépend de l'environnement utilisé. S'il s'agit du JDK de SUN, il vous suffira d'utiliser successivement ces commandes :

```
javac Point.java
javac TstPoint.java
java TstPoint
```

Avec un environnement de développement intégré, les choses se dérouleront de façon plus ou moins automatique. Souvent, il vous suffira de définir un "fichier projet" contenant simplement les noms des fichiers source concernés (*Point.java* et *TstPoint.java*).

1.3.2 Plusieurs classes dans un même fichier source

Jusqu'ici, nous avons :

- déclaré chaque classe avec l'attribut *public* (ne confondez pas ce droit d'accès à une classe avec le droit d'accès à ses champs ou méthodes, même si certains mots-clés sont communs) ;
- placé une seule classe par fichier source.

En fait, Java n'est pas tout à fait aussi strict. Il vous impose seulement de respecter les contraintes suivantes :

- un fichier source peut contenir plusieurs classes mais une seule doit être publique ;
- la classe contenant la méthode *main* doit obligatoirement être publique, afin que la machine virtuelle y ait accès ;
- une classe n'ayant aucun attribut d'accès[1] reste accessible à toutes les classes du même paquetage donc, a fortiori, du même fichier source.

Ainsi, tant que nous ne cherchons pas à utiliser *Point* en dehors de *TstPoint*, nous pouvons regrouper ces deux classes *TstPoint* et *Point* à l'intérieur d'un seul fichier, en procédant ainsi (nous avons changé le nom de la classe *TstPoint* en *TstPnt2*) :

```
public class TstPnt2
{ public static void main (String args[])
  { Point a ;
    a = new Point() ;
    a.initialise(3, 5) ;
    a.affiche() ;
    a.deplace(2, 0) ;
    a.affiche() ;
    Point b = new Point() ;
    b.initialise (6, 8) ;   b.affiche() ;
  }
}
```

1. On verra que l'attribut d'accès d'une classe ne peut prendre que deux valeurs : inexistant (accès de paquetage) ou *public*.

```
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 un point de coordonnees " + x + " " + y) ;
  }
  private int x ;   // abscisse
  private int y ;   // ordonnee
}
```

Les classes TstPnt2 *et* Point *dans un seul fichier source*

Par souci de clarté, il nous arrivera souvent de fournir des exemples de programmes complets sous cette forme même si, en pratique, vous serez généralement amené à dissocier vos classes et à les rendre publiques pour pouvoir les réutiliser le plus largement possible.

Remarques

1 Vous voyez que la classe *Clavier* proposée au paragraphe 4.1 du chapitre 2 peut être exploitée de cette manière. Autrement dit, il suffit de la recopier dans tout programme y faisant appel, en supprimant simplement le mot-clé *public*.

2 En théorie, rien ne vous empêche de faire de la méthode *main* une méthode de la classe *Point*. Il serait ainsi possible d'écrire un programme réduit à une seule classe et contenant à la fois sa définition et son utilisation. Mais dans ce cas, la méthode *main* aurait accès aux champs privés de la classe, ce qui ne correspond pas aux conditions usuelles d'utilisation :

```
class Point
  { // méthodes initialise, deplace et affiche
  public static void main (String args[])
  { Point a ;
    a = new Point() ;
     .....
    a.x = ...      // autorisé ici puisque main est une méthode de Point
  }
  private int x, y ;
}
```

Il n'est donc pas judicieux d'utiliser cette possibilité pour faire de la méthode *main* une sorte de test de la classe, même si cette démarche est parfois utilisée dans la littérature sur Java.

3 Lorsque plusieurs classes figurent dans un même fichier source, la compilation crée un fichier *.class* par classe.

2 La notion de constructeur

2.1 Généralités

Dans l'exemple de classe *Point* du paragraphe 1.1, il est nécessaire de recourir à la méthode *initialise* pour attribuer des valeurs aux champs d'un objet de type *Point*. Une telle démarche suppose que l'utilisateur de l'objet fera effectivement l'appel voulu au moment opportun. En fait, la notion de constructeur vous permet d'automatiser le mécanisme d'initialisation d'un objet. En outre, cette initialisation ne sera pas limitée à la mise en place de valeurs initiales ; il pourra s'agir de n'importe quelles actions utiles au bon fonctionnement de l'objet.

Un constructeur n'est rien d'autre qu'une méthode, sans valeur de retour, portant le même nom que la classe. Il peut disposer d'un nombre quelconque d'arguments (éventuellement aucun).

2.2 Exemple de classe comportant un constructeur

Considérons la classe *Point* présentée au paragraphe 1.1 et transformons simplement la méthode *initialise* en un constructeur en la nommant *Point*. La définition de notre nouvelle classe se présente alors ainsi :

```
public class Point
{ public Point (int abs, int ord)  // constructeur
  { x = abs ;
    y = ord ;
  }
  public void deplace (int dx, int dy)
  { x += dx ;
    y += dy ;
  }
  public void affiche ()
  { System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;
  }
  private int x ;   // abscisse
  private int y ;   // ordonnee
}
```

Définition d'une classe Point *munie d'un constructeur*

Comment utiliser cette classe ? Cette fois, une instruction telle que :

```
Point a = new Point() ;
```

ne convient plus : elle serait refusée par le compilateur. En effet, à partir du moment où une classe dispose d'un constructeur, il n'est plus possible de créer un objet sans l'appeler. Ici, notre constructeur a besoin de deux arguments. Ceux-ci doivent obligatoirement être fournis lors de la création, par exemple :

```
Point a = new Point(1, 3) ;
```

À titre d'exemple, voici comment on pourrait adapter le programme du paragraphe 1.3 en utilisant cette nouvelle classe *Point* :

```
public class TstPnt3
{ public static void main (String args[])
  { Point a ;
    a = new Point(3, 5) ;
    a.affiche() ;
    a.deplace(2, 0) ;
    a.affiche() ;
    Point b = new Point(6, 8) ;
    b.affiche() ;
  }
}

class Point
{ public Point (int abs, int ord)  // constructeur
  { x = abs ;
    y = ord ;
  }
  public void deplace (int dx, int dy)
  { x += dx ;
    y += dy ;
  }
  public void affiche ()
  { System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;
  }
  private int x ;   // abscisse
  private int y ;   // ordonnee
}
```

```
Je suis un point de coordonnees 3 5
Je suis un point de coordonnees 5 5
Je suis un point de coordonnees 6 8
```

Exemple d'utilisation d'une classe Point *munie d'un constructeur*

2.3 Quelques règles concernant les constructeurs

1 Par essence, un constructeur ne fournit aucune valeur. Dans son en-tête, aucun type ne doit figurer devant son nom. Même la présence (logique) de *void* est une erreur :

```
class Truc
{ .....
  public void Truc ()   // erreur de compilation : void interdit ici
  { .....
  }
}
```

2 Une classe peut ne disposer d'aucun constructeur (c'était le cas de notre première classe *Point*). On peut alors instancier des objets comme s'il existait un constructeur par défaut sans arguments (et ne faisant rien) par des instructions telles que :

```
Point a =  new Point() ;   // OK si Point n'a pas de constructeur
```

Mais dès qu'une classe possède au moins un constructeur (nous verrons plus loin qu'elle peut en comporter plusieurs), ce pseudo-constructeur par défaut ne peut plus être utilisé, comme le montre cet exemple :

```
class A
{ public A(int) { ..... }   // constructeur à un argument int
  .....
}
   .....
A a1 = new A(5) ; // OK
A a2 = new A() ;  // erreur
```

On notera que l'emploi d'un constructeur sans arguments ne se distingue pas de celui du constructeur par défaut. Si, pour une classe *T* donnée, l'instruction suivante est acceptée :

```
T t = new T() ;
```

cela signifie simplement que :

- soit *T* ne dispose d'aucun constructeur,
- soit *T* dispose d'un constructeur sans arguments.

3 Un constructeur ne peut pas être appelé directement depuis une autre méthode. Par exemple, si *Point* dispose d'un constructeur à deux arguments de type *int* :

```
Point a = new Point (3, 5) ;
   .....
a.Point(8, 3) ;  // interdit
```

4 Un constructeur peut appeler un autre constructeur de la même classe. Cette possibilité utilise la surdéfinition des méthodes et nécessite l'utilisation du mot-clé *super* ; nous en parlerons plus loin.

5 Un constructeur peut être déclaré privé (*private*). Dans ce cas, il ne pourra plus être appelé de l'extérieur, c'est-à-dire qu'il ne pourra pas être utilisé pour instancier des objets :

```
class A
{ private A() { ..... } // constructeur privé sans arguments
  .....
}
.....
A a() ;     // erreur : le constructeur correspondant A() est privé[1]
```

En fait, cette possibilité n'aura d'intérêt que si la classe possède au moins un autre constructeur public faisant appel à ce constructeur privé, qui apparaîtra alors comme une méthode de service.

2.4 Construction et initialisation d'un objet

Dans nos précédents exemples, le constructeur initialisait les différents champs privés de l'objet. En fait, contrairement à ce qui se passe pour les variables locales, les champs d'un objet sont toujours initialisés par défaut. En outre, il est possible de leur attribuer explicitement une valeur au moment de leur déclaration. En définitive, la création d'un objet entraîne toujours, par ordre chronologique, les opérations suivantes :

- une initialisation par défaut de tous les champs de l'objet,
- une initialisation explicite lors de la déclaration du champ,
- l'exécution des instructions du corps du constructeur.

2.4.1 Initialisation par défaut des champs d'un objet

Dès qu'un objet est créé, et avant l'appel du constructeur, ses champs sont initialisés à une valeur par défaut "nulle" ainsi définie :

Type du champ	Valeur par défaut
boolean	false
char	caractère de code nul
entier (byte, short, int, long)	0
flottant (float, double)	0.f ou 0.
objet	null

Initialisation par défaut des champs d'un objet

Comme on peut s'y attendre, un champ d'une classe peut très bien être la référence à un objet. Dans ce cas, cette référence est initialisée à une valeur conventionnelle notée *null* sur laquelle nous reviendrons.

2.4.2 Initialisation explicite des champs d'un objet

Une variable locale peut être initialisée lors de sa déclaration. Il en va de même pour un champ. Considérons :

1. N'oubliez pas que dès qu'une classe dispose d'un constructeur, on ne peut plus recourir au pseudo-constructeur par défaut.

```
class A
{  public A (...) { ..... }   //  constructeur de A
      .....
   private int n = 10 ;
   private int p ;
}
```

L'instruction suivante :

```
A a = new A (...) ;
```

entraîne successivement :

- l'initialisation (implicite) des champs *n* et *p* de *a* à 0,
- l'initialisation (explicite) du champ *n* à la valeur figurant dans sa déclaration, soit 10,
- l'exécution des instructions du constructeur.

Comme celle d'une variable locale, une initialisation de champ peut théoriquement comporter non seulement une constante ou une expression constante, mais également n'importe quelle expression (pour peu qu'elle soit calculable au moment voulu). En voici un exemple :

```
class B
{ .....
  private int n = 10 ;
  private int p = n+2 ;
}
```

Notez que si l'on inverse l'ordre des déclarations de *n* et de *p*, on obtient une erreur de compilation car *n* n'est pas encore connu lorsqu'on rencontre l'expression d'initialisation *n+2* :

```
class B
{ .....
  private int p = n+2 ;   // erreur : n n'est pas encore connu ici
  private int n = 10 ;
}
```

En général, il n'est guère prudent d'utiliser ce genre de possibilité car elle ne permet pas un réarrangement des déclarations de champs. De même, il n'est guère raisonnable d'initialiser un champ de cette manière, pourtant acceptée par Java :

```
class C
{ .....
  private int n = Clavier.lireInt() ;
}
```

2.4.3 Appel du constructeur

Le corps du constructeur n'est exécuté qu'après l'initialisation par défaut et l'initialisation explicite. Voici un exemple d'école dans lequel cet ordre a de l'importance :

```
public class Init
{ public static void main (String args[])
  { A a = new A() ;   // ici a.n vaut 5, a.p vaut 10, mais a.np vaut 200
    a.affiche() ;
  }
}
```

```
class A
{ public A()
  { // ici, n vaut 20, p vaut 10 et np vaut 0
    np = n * p ;
    n = 5 ;
  }
  public void affiche()
  { System.out.println ("n = " + n + ", p = " + p + ", np = " + np) ;
  }
  private int n = 20, p = 10 ;
  private int np ;
}
```

```
n = 5, p = 10, np = 200
```

Quand l'ordre des différentes initialisations a de l'importance

En pratique, on aura intérêt à s'arranger pour que l'utilisateur de la classe n'ait pas à s'interroger sur l'ordre chronologique exact de ces différentes opérations. Autrement dit, dans la mesure du possible, on veillera à ne pas mêler les différentes possibilités d'initialisation d'un même champ et à limiter les dépendances. Ici, il serait beaucoup plus simple de procéder ainsi :

```
class A
{ public A()
  { n = 5 ; p = 10 ;
    np = n * p ;
  }
  .....
}
```

D'une manière générale, les possibilités offertes par le constructeur sont beaucoup plus larges que les initialisations explicites, ne serait-ce que parce que seul le constructeur peut récupérer les arguments fournis à *new*. Sauf cas particulier, il est donc préférable d'effectuer les initialisations dans le constructeur.

2.4.4 Cas des champs déclarés avec l'attribut final

Nous avons déjà vu qu'on pouvait déclarer une variable locale avec l'attribut *final*. Dans ce cas, sa valeur ne devait être définie qu'une seule fois. Cette possibilité se transpose aux champs d'un objet. Examinons quelques exemples, avant de dégager les règles générales.

Exemple 1

```
class A
{ .....
  private final int n = 20 ;  // la valeur de n est définie dans sa déclaration
  .....
}
```

Ici, la valeur de *n* est une constante fixée dans sa déclaration. Notez que tous les objets de type *A* posséderont un champ *n* contenant la valeur 10 (nous verrons plus loin qu'il serait plus pratique et plus économique de faire de *n* un champ de classe en le déclarant avec l'attribut *static*).

Exemple 2

```
class A
{ public A()
  { n = 10 ;
  }
  private final int n ;
}
```

Ici, la valeur de *n* est définie par le constructeur de *A*. On a affaire à une initialisation tardive, comme pour une variable locale.

Ici encore, telle que la classe *A* a été définie, tous les objets de type *A* auront un champ *n* comportant la même valeur.

Exemple 3

Considérez maintenant :

```
class A
{ public A(int nn)
  { n = nn ;
  }
  private final int n ;
}
```

Cette fois, les différents objets de type *A* pourront posséder des valeurs de *n* différentes.

Quelques règles

Comme une variable locale, un champ peut donc être déclaré avec l'attribut *final*, afin d'imposer qu'il ne soit initialisé qu'une seule fois. Toute tentative de modification ultérieure conduira à une erreur de compilation. Mais, alors qu'une variable locale pouvait être initialisée tardivement n'importe où dans une méthode, un champ déclaré *final* doit être initialisé au plus tard par un constructeur (ce qui est une bonne précaution).

D'autre part, il n'est pas permis de compter sur l'initialisation par défaut d'un tel champ. Le schéma suivant conduira à une erreur de compilation :

```
class A
{ A()
  { // ici, on ne donne pas de valeur à n
  }
  private final int n ; // ici, non plus --> erreur de compilation
}
```

Remarques

1 Le champ *final* de notre exemple était privé mais, bien entendu, il pourraît être public.

2 Nous verrons que le mot-clé *final* peut s'appliquer à une méthode avec une signification totalement différente.

Informations complémentaires

Outre les possibilités d'initialisation explicite des champs, Java permet d'introduire, dans la définition d'une classe, un ou plusieurs blocs d'instructions dits "blocs d'initialisation". Ils seront exécutés dans l'ordre où ils apparaissent après les initialisations explicites et avant l'appel du constructeur. Cette possibilité n'est pas indispensable (on peut faire la même chose dans un constructeur !). Elle risque même de nuire à la clarté du programme. Son usage est déconseillé.

3 Éléments de conception des classes

Ce paragraphe vous propose quelques éléments fondamentaux pour la bonne conception de vos classes.

3.1 Les notions de contrat et d'implémentation

L'encapsulation des données n'est pas obligatoire en Java. Il est cependant vivement conseillé d'y recourir systématiquement en déclarant tous les champs privés.

En effet, une bonne conception orientée objets s'appuie généralement sur la notion de *contrat*, qui revient à considérer qu'une classe est caractérisée par un ensemble de services définis par :

- les en-têtes de ses méthodes publiques[1],
- le comportement de ces méthodes.

Le reste, c'est-à-dire les champs et les méthodes privés ainsi que le corps des méthodes publiques, n'a pas à être connu de l'utilisateur de la classe. Il constitue ce que l'on appelle souvent l'*implémentation* de la classe.

En quelque sorte, le contrat définit ce que fait la classe tandis que son implémentation précise comment elle le fait.

Il est clair que le grand mérite de l'encapsulation des données (*private*) est de permettre au concepteur d'une classe d'en modifier l'implémentation, sans que l'utilisateur n'ait à modifier les programmes qui l'exploitent.

1. Dans certains langages, on parle d'interface, mais en Java ce terme possède une autre signification.

On notera cependant que les choses ne seront entièrement satisfaisantes que si le contrat initial est respecté. Or s'il est facile de respecter les en-têtes de méthodes, il peut en aller différemment en ce qui concerne leur comportement. En effet, ce dernier n'est pas inscrit dans le code de la classe elle-même, mais simplement spécifié par le concepteur[1]. Il va de soi que tout dépend alors de la qualité de sa spécification et de sa programmation.

3.2 Typologie des méthodes d'une classe

Parmi les différentes méthodes que comporte une classe, on a souvent tendance à distinguer :

- les constructeurs ;
- les méthodes d'accès (en anglais *accessor*) qui fournissent des informations relatives à l'état d'un objet, c'est-à-dire aux valeurs de certains de ses champs (généralement privés), sans les modifier ;
- les méthodes d'altération (en anglais *mutator*) qui modifient l'état d'un objet, donc les valeurs de certains de ses champs.

On rencontre souvent l'utilisation de noms de la forme *getXXXX* pour les méthodes d'accès et *setXXXX* pour les méthodes d'altération, y compris dans des programmes dans lesquels les noms de variables sont francisés. Par exemple, la classe *Point* du paragraphe annexe 2.2 pourrait être complétée par les méthodes suivantes :

```
public int getX { return x ; }            // getX ou encore getAbscisse
public int getY { return y ; }            // getY ou encore getOrdonnee
public void setX (int abs) { x = abs ; }  // setX ou encore setAbscisse
public void setY (int ord) { x = ord ; }  // setY ou encore setOrdonnee

public void setPosition (int abs, int ord)
{ x = abs ; y = ord ;
}
```

Notez qu'il n'est pas toujours prudent de prévoir une méthode d'altération pour chacun des champs privés d'un objet. En effet, il ne faut pas oublier qu'il doit toujours être possible de modifier l'implémentation d'une classe de manière transparente pour son utilisateur. Même sur les petits exemples précédents, des difficultés pourraient apparaître si nous souhaitions représenter un point (de façon privée), non plus par ses coordonnées cartésiennes, mais par ses coordonnées polaires. Dans ce cas, en effet, la méthode *setX* ne serait plus utilisable seule ; elle ne pourrait l'être que conjointement à *setY*. Il pourrait alors être préférable de ne conserver que la méthode *setPosition*.

1. Imaginez une méthode *deplace* implémentée ainsi :
void deplace (int dx, int dy) { x +=dy ; y+=dx; }

4 Affectation et comparaison d'objets

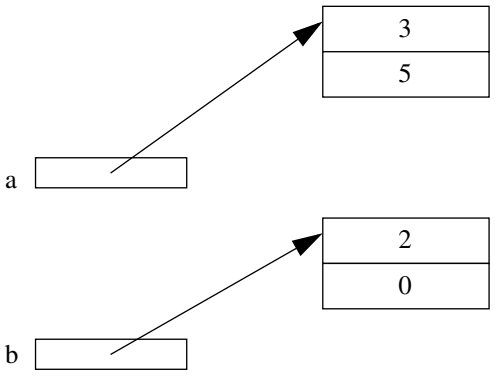
Nous avons étudié le rôle de l'opérateur d'affectation sur des variables d'un type primitif. Par ailleurs, nous venons de voir qu'il existe des variables de type classe, destinées à contenir des références sur des objets. Comme on peut s'y attendre, ces variables pourront être soumises à des affectations. Mais celles-ci portent sur les références et non sur les objets eux-mêmes, ce qui modifie quelque peu la sémantique (signification) de l'affectation. C'est ce que nous allons examiner à partir de deux exemples. Nous donnerons ensuite quelques informations concernant l'initialisation de références. Enfin, nous montrerons le rôle des opérateurs == et != lorsqu'on les applique à des références.

4.1 Premier exemple

Supposons que nous disposions d'une classe *Point* possédant un constructeur à deux arguments entiers et considérons ces instructions :

```
Point a, b ;
   .....
a = new Point (3, 5) ;
b = new Point (2, 0) ;
```

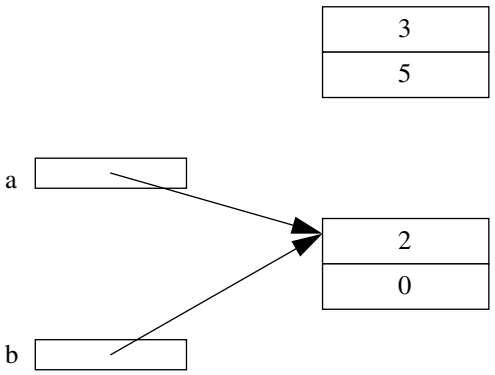
Après leur exécution, on aboutit à cette situation :



Exécutons maintenant l'affectation :

```
a = b ;
```

Celle-ci recopie simplement dans *a* la référence contenue dans *b*, ce qui nous conduit à :



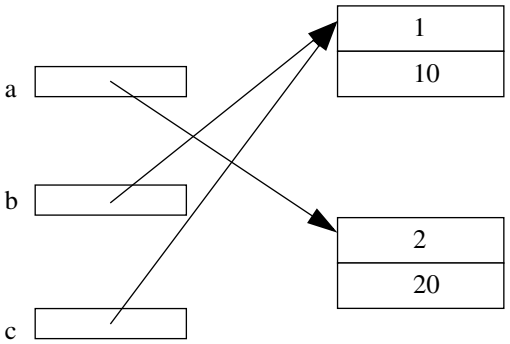
Dorénavant, *a* et *b* désignent le même objet, et non pas deux objets de même valeur.

4.2 Second exemple

Considérons les instructions suivantes :

```
Point a, b, c ;
   .....
a = new Point (1, 10) ;
b = new Point (2, 20) ;
c = a ;
a = b ;
b = c ;
```

Après leur exécution, on aboutit à cette situation :



Notez bien qu'il n'existe ici que deux objets de type *Point* et trois variables de type *Point* (trois références, dont deux de même valeur).

Remarque

Le fait qu'une variable de type classe soit une référence et non une valeur aura aussi des conséquences dans la transmission d'un objet en argument d'une méthode.

4.3 Initialisation de référence et référence nulle

Nous avons déjà vu qu'il n'est pas possible de définir une variable locale d'un type primitif sans l'initialiser. La règle se généralise aux **variables locales de type classe**. Considérez cet exemple utilisant une classe *Point* disposant d'une méthode *affiche* :

```
public static void main (String args[])
{ Point p ;       // p est locale à main
  p.affiche() ;   // erreur de compilation : p n'a pas encore reçu de valeur
   .....
}
```

En revanche, comme nous l'avons vu au paragraphe 2.4, un champ d'un objet est toujours initialisé soit implicitement à une valeur dite *nulle*[1], soit explicitement, soit au sein du constructeur. Cette règle s'applique également aux champs de type classe[2], pour lesquels cette valeur nulle correspond à une valeur particulière de référence notée par le mot-clé *null*.

Conventionnellement, une telle référence ne désigne aucun objet. Elle peut être utilisée dans une comparaison, comme dans cet exemple (on suppose que *Point* est une classe) :

```
class A
{ public void f()
  { .....
    if (p==nul) .....  // on compare la valeur de p à la valeur null
  }
  private Point p ;
}
```

La valeur *null* peut aussi être affectée explicitement à une variable ou un champ de type classe. En général, cela ne présentera guère d'intérêt. En tout cas, il ne faut pas se reposer là-dessus pour éviter de tester une référence qui risque de ne pas être définie ou d'être nulle. En effet, alors qu'une référence non définie est détectée en compilation, une référence nulle n'est détectée qu'au moment où l'on cherche à l'employer pour lui appliquer une méthode, donc à l'exécution. On obtient une exception *NullPointerException* qui, si elle n'est pas traitée (comme nous apprendrons à le faire au chapitre 10, conduit à un arrêt de l'exécution. Voyez cet exemple :

1. Exception faite des champs déclarés avec l'attribut *final* qui, comme on l'a vu, doivent recevoir explicitement une valeur.

2. Nous en verrons des exemples au paragraphe 11.

```
public static void main (String args[])
{ Point p = null ;  // p est locale à main et initialisée à null
  p.affiche() ;     // erreur d'exécution cette fois
   .....
}
```

4.4 La notion de clone

Nous venons de voir que l'affectation de variables de type objet se limite à la recopie de références. Elle ne provoque pas la recopie de la valeur des objets. Si on le souhaite, on peut bien entendu effectuer explicitement la recopie de tous les champs d'un objet dans un autre objet de même type. Toutefois, si les données sont convenablement encapsulées, il n'est pas possible d'y accéder directement. On peut songer à s'appuyer sur l'existence de méthodes d'accès et d'altération de ces champs privés. Cependant, rien ne permet d'être certain que ces méthodes forment un ensemble cohérent et complet (nous avons déjà évoqué au paragraphe 3.2 les difficultés à concilier complétude et transparence de l'implémentation). Quand bien même ce serait le cas, leur utilisation pour la recopie complète d'un objet nécessiterait malgré tout une bonne connaissance de son implémentation.

En fait, la démarche la plus réaliste consiste plutôt à prévoir dans la classe correspondante une méthode destinée à fournir une copie de l'objet concerné, comme dans cet exemple[1] :

```
class Point
{ public Point(int abs, int ord)  { x = abs ; y = ord ;  }
  public Point copie ()       // renvoie une référence à un Point
  { Point p = new Point(x, y) ;
    p.x = x ; p.y = y ;
    return p ;
  }
  private int x, y ;
}
   .....
Point a = new Point(1, 2) ;
Point b = a.copie() ;       // b est une copie conforme de a
```

Cette démarche est utilisable tant que la classe concernée ne comporte pas de champs de type classe. Dans ce cas, il faut décider si leur copie doit, à son tour, porter sur les objets référencés plutôt que sur les références.

On voit apparaître la distinction usuelle entre :

- *la copie superficielle* d'un objet : on se contente de recopier la valeur de tous ses champs, y compris ceux de type classe,

1. Nous reviendrons plus loin sur la possibilité pour une méthode de renvoyer une référence à un objet et nous verrons qu'aucun problème particulier ne se pose (contrairement à ce qui se passe dans d'autres langages comme C++).

- *la copie profonde* d'un objet : comme précédemment, on recopie la valeur des champs d'un type primitif mais pour les champs de type classe, on crée une nouvelle référence à un autre objet du même type de même valeur.

Comme on s'en doute, la copie profonde peut être récursive et pour être menée à bien, elle demande la connaissance de la structure des objets concernés.

La démarche la plus rationnelle pour traiter cette copie profonde qu'on nomme *clonage* en Java, consiste à faire en sorte que chaque classe concernée par l'éventuelle récursion dispose de sa propre méthode.

En C++

En C++, l'affectation réalise une copie superficielle des objets (rappelons qu'il ne s'agit pas, comme en Java, d'une copie de références). On peut redéfinir l'opérateur d'affectation (pour une classe donnée) et lui donner la signification de son choix ; en général, on le transforme en une copie profonde.

En outre, il existe en C++ un constructeur particulier dit *constructeur par recopie* qui joue un rôle important dans les transmissions d'objets en argument d'une méthode. Par défaut, il effectue lui aussi une copie superficielle ; il peut également être redéfini pour réaliser une copie profonde.

4.5 Comparaison d'objets

Les opérateurs == et != s'appliquent théoriquement à des objets. Mais comme ils portent sur les références elles-mêmes, leur intérêt est très limité. Ainsi, avec :

```
Point a, b ;
```

L'expression *a* == *b* est vraie uniquement si *a* et *b* font référence à un seul et même objet, et non pas seulement si les valeurs des champs de *a* et *b* sont les mêmes.

5 Le ramasse-miettes

Nous avons vu comment un programme peut donner naissance à un objet en recourant à l'opérateur *new*[1]. À sa rencontre, Java alloue un emplacement mémoire pour l'objet et l'initialise (implicitement, explicitement, par le constructeur).

En revanche, il n'existe aucun opérateur permettant de détruire un objet dont on n'aurait plus besoin.

1. Il peut s'agir d'un recours indirect comme dans *a.copie()*.

En fait, la démarche employée par Java est un mécanisme de gestion automatique de la mémoire connu sous le nom de *ramassse-miettes* (en anglais *Garbage Collector*). Son principe est le suivant :

- À tout instant, on connaît le nombre de références à un objet donné. On notera que cela n'est possible que parce que Java gère toujours un objet par référence.
- Lorsqu'il n'existe plus aucune référence sur un objet, on est certain que le programme ne pourra plus y accéder. Il est donc possible de libérer l'emplacement correspondant, qui pourra être utilisé pour autre chose. Cependant, pour des questions d'efficacité, Java n'impose pas que ce travail de récupération se fasse immédiatement. En fait, on dit que l'objet est devenu *candidat au ramasse-miettes*.

Remarque

On peut créer un objet sans en conserver la référence, comme dans cet exemple artificiel :

```
(new Point(3,5)).affiche() ;
```

Ici, on crée un objet dont on affiche les coordonnées. Dès la fin de l'instruction, l'objet (qui n'est pas référencé) devient candidat au ramasse-miettes.

En C++

L'opérateur *delete* permet de détruire un objet (dynamique) créé par *new*. Les objets automatiques sont automatiquement détruits lors de la sortie du bloc correspondant. La destruction d'un objet (dynamique ou automatique) entraîne l'appel d'une méthode particulière dite destructeur. Il n'existe pas de ramasse-miettes en C++.

Informations complémentaires

Avant qu'un objet soit soumis au ramasse-miettes, Java appelle la méthode *finalize* de sa classe[1]. En théorie, on pourrait se fonder sur cet appel pour libérer des ressources qui ne le seraient pas automatiquement, comme des fichiers ouverts, des allocations de mémoire, des éléments verrouillés... En pratique, cependant, on est fortement limité par le fait qu'on ne maîtrise pas le moment de cet appel. Dans bon nombre de cas d'ailleurs, le ramasse-miettes ne se déclenche que lorsque la mémoire commence à se faire rare...

1. Nous verrons plus tard que toute classe dispose toujours d'une méthode *finalize* par défaut qu'elle hérite de la super-classe *Object* mais qu'il est possible d'y redéfinir cette méthode.

6 Règles d'écriture des méthodes

Jusqu'ici, nous nous sommes contenté de dire qu'une méthode était formée d'un bloc précédé d'un en-tête. Nous allons maintenant apporter quelques précisions concernant les règles d'écriture d'une méthode, ce qui nous permettra de distinguer les méthodes fonctions des autres et d'aborder les arguments muets, les arguments effectifs et leurs éventuelles conversions, et enfin les variables locales.

6.1 Méthodes fonction

Une méthode peut ne fournir aucun résultat. Le mot-clé *void* figure alors dans son en-tête à la place du type de la valeur de retour. Nous avons déjà rencontré des exemples de telles méthodes dont l'appel se présente sous la forme :

Objet.méthode (liste d'arguments)

Mais une méthode peut aussi fournir un résultat. Nous parlerons alors de méthode fonction. Voici par exemple une méthode *distance* qu'on pourrait ajouter à une classe *Point* pour obtenir la distance d'un point à l'origine :

```
public class Point
{    .....
   double distance ()
   { double d ;
     d = Math.sqrt (x*x* + y*y) ;
     return d ;
   }
   private int x, y ;
     .....
}
```

De même, voici deux méthodes simples (déjà évoquées précédemment) permettant d'obtenir l'abscisse et l'ordonnée d'un point :

```
int getX { return x ; }
int getY { return y ; }
```

La valeur fournie par une méthode fonction peut apparaître dans une expression, comme dans ces exemples utilisant les méthodes *distance*, *getX* et *getY* précédentes :

```
Point a = new Point(...) ;
double u, r ;
   .....
u = 2. * a.distance() ;
r = Math.sqrt(a.getX() * a.getX() + a.getY() * a.getY() ) ;
```

On peut ne pas utiliser la valeur de retour d'une méthode. Par exemple, cette instruction est correcte (même si, ici, elle ne sert à rien) :

```
a.distance() ;
```

Bien entendu, cette possibilité n'aura d'intérêt que si la méthode fait autre chose que de calculer une valeur.

6.2 Les arguments d'une méthode

6.2.1 Arguments muets ou effectifs

Comme dans tous les langages, les arguments figurant dans l'en-tête de la définition d'une méthode se nomment *arguments muets* (ou encore arguments ou paramètres formels). Ils jouent un rôle voisin de celui d'une variable locale à la méthode, avec cette seule différence que leur valeur sera fournie à la méthode au moment de son appel. Par essence, ces arguments sont de simples identificateurs ; il serait absurde de vouloir en faire des expressions.

Il est possible de déclarer un argument muet avec l'attribut *final*. Dans ce cas, le compilateur s'assure que sa valeur n'est pas modifiée par la méthode :

```
void f (final int n, double x)
{  .....
   n = 12 ;    // erreur de compilation
   x = 2.5 ;   // OK
   .....
}
```

Les arguments fournis lors de l'appel de la méthode portent quant à eux le nom d'*arguments effectifs* (ou encore paramètres effectifs). Comme on l'a déjà vu à travers de nombreux exemples, en Java, il peut s'agir d'expressions (bien sûr, un simple nom de variable ou une constante constituent des cas particuliers d'expressions). On notera que cela n'est possible que parce que ce sont les valeurs de ces arguments qui seront effectivement transmises[1] ; nous reviendrons en détail sur ce point au paragraphe 9.

6.2.2 Conversion des arguments effectifs

Jusqu'ici, nous avions appelé nos différentes méthodes en utilisant des arguments effectifs d'un type identique à celui de l'argument muet correspondant. En fait, Java fait preuve d'une certaine tolérance en vous permettant d'utiliser un type différent. Il faut simplement que la conversion dans le type attendu soit une conversion implicite légale, autrement dit qu'elle respecte la hiérarchie (ou encore, ce qui revient au même, qu'il s'agisse d'une conversion autorisée par affectation).

Voici quelques exemples usuels :

```
class Point
{ .....
  void deplace (int dx, int dy) { ..... }
  .....
}
   .....
Point p = new Point(...) ;
int n1, n2 ; byte b ; long q ;
   .....
```

1. Dans les langages où la transmission des arguments se fait par adresse (ou par référence), les arguments effectifs ne peuvent pas être des expressions.

```
p.deplace (n1, n2) ;     // OK : appel normal
p.deplace (b+3, n1) ;    // OK : b+3 est déjà de type int
p.deplace (b, n1) ;      // OK : b de type byte sera converti en int
p.deplace (n1, q) ;      // erreur : q de type long ne peut être converti en int
p.deplace (n1, (int)q) ; // OK
```

Voici quelques autres exemples plus insidieux :

```
class Point
{ .....
  void deplace (byte dx, byte dy) { ..... }
  .....
}
   .....
Point p = new Point(...) ;
byte b1, b2 ;
   .....
p.deplace (b1, b2) ;   // OK : appel normal
p.deplace (b1+1, b2) ; // erreur : b1+1 de type int ne peut être converti en byte
p.deplace (b1++, b2) ; // OK : b1++ est de type byte
                       //  (mais peu conseillé : on a modifié la valeur de b1)
```

Informations complémentaires

En général, il n'est pas utile de savoir dans quel ordre sont évaluées les expressions figurant en arguments effectifs d'un appel de méthode. Considérez cependant :

```
int n=5 ; double a=2.5, b=3.5 ;
f (n++, n, a=b, a)
```

Le premier argument a pour valeur 5 (*n* avant incrémentation). En revanche, la valeur du deuxième dépend de son ordre de calcul par rapport au précédent. Comme **Java respecte l'ordre des arguments**, on voit qu'il vaudra 6 (valeur de *n* à ce moment-là). Le troisième argument qui correspond à la valeur de l'affectation *a=b* vaudra 3.5. Le dernier vaudra également 3.5 puisqu'il s'agit de la valeur de *a*, après les évaluations précédentes. En revanche, en remplaçant l'appel précédent par :

```
f (n, n++, a, a=b)
```

les valeurs des arguments seront respectivement 5, 5, 2.5 et 3.5.

Notez qu'il n'est pas prudent d'écrire des programmes fondés sur ces règles d'évaluation. D'ailleurs, dans de nombreux langages (C, C++ notamment), aucun ordre précis n'est prévu dans de telles situations.

6.3 Propriétés des variables locales

Ce paragraphe fait le point sur les variables locales, que nous avons déjà utilisées de manière plus ou moins intuitive. Il reprend donc un certain nombre d'informations qui ont déjà été exposées au fil de l'ouvrage.

Comme on s'en doute, la portée d'une variable locale (emplacement du source où elle est accessible) est limitée au bloc constituant la méthode où elle est déclarée. De plus, une variable locale ne doit pas posséder le même nom qu'un argument muet de la méthode[1] :

```
void f(int n)
{ float x ;      // variable locale à f
  float n ;      // interdit en Java
  .....
}
void g ()
{ double x ;    // variable locale à g, indépendante de x locale à f
    .....
}
```

L'emplacement d'une variable locale est alloué au moment où l'on entre dans la fonction et il est libéré lorsqu'on en sort. Cela signifie bien sûr que cet emplacement peut varier d'un appel au suivant ce qui, en Java, n'a guère d'importances en pratique. Mais cela signifie surtout que les valeurs des variables locales ne sont pas conservées d'un appel au suivant ; on dit qu'elles ne sont pas *rémanentes*[2].

Notez bien que les variables définies dans la méthode *main* sont aussi des variables locales. Elles ont cependant cette particularité de n'être allouées qu'une seule fois (avant le début de *main*) et d'exister pendant toute la durée du programme (ou presque). Leur caractère non rémanent n'a plus alors aucune incidence.

Une variable locale est obligatoirement d'un type primitif ou d'un type classe ; dans ce dernier cas, elle contient la référence à un objet. On notera qu'il n'existe pas d'objets locaux à proprement parler, mais seulement des références locales à des objets dont l'emplacement mémoire est alloué explicitement par un appel à *new*.

Comme nous l'avons déjà mentionné, les variables locales ne sont pas initialisées de façon implicite (contrairement aux champs des objets). Toute variable locale, y compris une référence à un objet, doit être initialisée avant d'être utilisée, faute de quoi on obtient une erreur de compilation.

Remarque

Les variables locales à une méthode sont en fait des variables locales au bloc constituant la méthode. En effet, on peut aussi définir des variables locales à un bloc. Dans ce cas, leur portée est tout naturellement limitée à ce bloc ; leur emplacement est alloué à l'entrée dans le bloc et il disparaît à la sortie. Il n'est pas permis qu'une variable locale porte le même nom qu'une variable locale d'un bloc englobant.

---

1. Dans certains langages, cette possibilité est autorisée, mais alors la variable locale masque l'argument muet de même nom. De toute façon, il s'agit d'une situation déconseillée.

2. D'ailleurs, sans cette propriété, le compilateur ne pourrait pas s'assurer de la bonne initialisation des variables locales.

```
void f()
{ int n ;        // n est accessible de tout le bloc constituant f
    .....
  for (...)
  { int p ;     // p n'est connue que dans le bloc de for
    int n ;     // interdit : n existe déjà dans un bloc englobant
      .....
  }
    .....
  { int p ;     // p n'est connue que dans ce bloc ; elle est allouée ici
    .....       // et n'a aucun rapport avec la variable p ci-dessus
  }             // et elle sera désallouée ici
    .....
}
```

Notez qu'on peut créer artificiellement un bloc, indépendamment d'une quelconque instruction structurée comme *if*, *for*. C'est le cas du deuxième bloc interne à notre fonction *f* ci-dessus.

7 Champs et méthodes de classe

En Java, on peut définir des champs qui, au lieu d'exister dans chacune des instances de la classe, n'existent qu'en un seul exemplaire pour toutes les instances d'une même classe. Il s'agit en quelque sorte de données globales partagées par toutes les instances d'une même classe. On parle alors de champs de classe ou de champs statiques. De même, on peut définir des méthodes de classe (ou statiques) qui peuvent être appelées indépendamment de tout objet de la classe (c'est le cas de la méthode *main*).

7.1 Champs de classe

7.1.1 Présentation

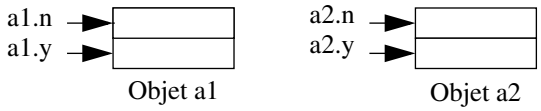
Considérons la définition (simpliste) de classe suivante (nous ne nous préoccuperons pas des droits d'accès aux champs *n* et *y*) :

```
class A
{ int n ;
  float y ;
}
```

Chaque objet de type *A* possède ses propres champs *n* et *x*. Par exemple, avec cette déclaration :

```
A a1 = new A(), a2 = new A() ;
```

on aboutit à une situation qu'on peut schématiser ainsi :

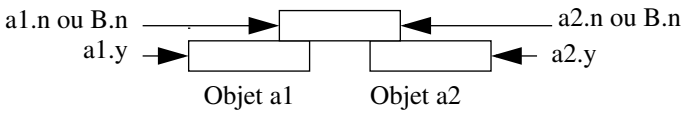


Mais Java permet de définir ce qu'on nomme des champs de classe (ou statiques) qui n'existent qu'en un seul exemplaire, quel que soit le nombre d'objets de la classe. Il suffit pour cela de les déclarer avec l'attribut *static*. Par exemple, si nous définissons :

```
class B
{ static int n ;
  float y ;
}

B a1 = new B(), a2 = new B() ;
```

nous aboutissons à cette situation :



Les notations *a1.n* et *a2.n* désignent donc le même champ. En fait, ce champ existe indépendamment de tout objet de sa classe. Il est possible (et même préférable) de s'y référer en le nommant simplement :

```
B.n       // champ (statique) n de la classe B
```

Bien entendu, ces trois notations (*a1.n*, *a2.n*, *B.n*) ne seront utilisables que pour un champ non privé. Il sera possible de prévoir des champs statiques privés, mais l'accès ne pourra alors se faire que par le biais de méthodes (nous verrons plus loin qu'il pourra s'agir de méthodes de classe).

Notez que depuis une méthode de la classe *B*, on accédera à ce champ on le nommant comme d'habitude *n* (le préfixe *B.* n'est pas nécessaire, mais il reste utilisable).

7.1.2 Exemple

Voici un exemple complet de programme utilisant une classe nommée *Obj* comportant un champ statique privé *nb*, destiné à contenir, à tout instant, le nombre d'objets de type *Obj* déjà créés. Sa valeur est incrémentée de 1 à chaque appel du constructeur. Nous nous contentons d'afficher sa valeur à chaque création d'un nouvel objet.

```
class Obj
{ public Obj()
  { System.out.print ("++ creation objet Obj ; ") ;
    nb ++ ;
    System.out.println ("il y en a maintenant " + nb) ;
  }
  private static long nb=0 ;
}
public class TstObj
{ public static void main (String args[])
  { Obj a ;
    System.out.println ("Main 1") ;
    a = new Obj() ;
    System.out.println ("Main 2") ;
    Obj b ;
    System.out.println ("Main 3") ;
    b = new Obj() ;
    Obj c = new Obj() ;
    System.out.println ("Main 4") ;
  }
}
```

```
Main 1
++ creation objet Obj ; il y en a maintenant 1
Main 2
Main 3
++ creation objet Obj ; il y en a maintenant 2
++ creation objet Obj ; il y en a maintenant 3
Main 4
```

Exemple d'utilisation d'un champ de classe

Remarque

La classe *Obj* ne tient pas compte des objets éventuellement détruits lors de l'exécution. En fait, on ne peut pas connaître le moment où un objet devient candidat au ramasse-miettes. En revanche, si son emplacement est récupéré, on sait qu'il y aura appel de la méthode *finalize*. En décrémentant le compteur d'objets de 1 dans cette méthode, on voit qu'on peut connaître plus précisément le nombre d'objets existant encore (y compris cependant ceux qui ne sont plus référencés mais pas encore détruits).

7.2 Méthodes de classe

7.2.1 Généralités

Nous venons de voir comment définir des champs de classe, lesquels n'existent qu'en un seul exemplaire, indépendamment de tout objet de la classe. De manière analogue, on peut imaginer que certaines méthodes d'une classe aient un rôle indépendant d'un quelconque objet. Ce serait notamment le cas d'une méthode se contentant d'agir sur des champs de classe ou de les utiliser.

Bien sûr, vous pouvez toujours appeler une telle méthode en la faisant porter artificiellement sur un objet de la classe (alors que la référence à un tel objet n'est pas utile). Là encore, Java vous permet de définir une *méthode de classe* en la déclarant avec le mot-clé *static*. L'appel d'une telle méthode ne nécessite plus que le nom de la classe correspondante.

Bien entendu, une méthode de classe ne pourra en aucun cas agir sur des champs usuels (non statiques) puisque, par nature, elle n'est liée à aucun objet en particulier. Voyez cet exemple :

```
class A
{   .....
  private float x ;        // champ usuel
  private static int n ;   // champ de classe
    .....
  public static void f()   // méthode de classe
  { .....     // ici, on ne peut pas accéder à x, champ usuel,
    .....     //  mais on peut accéder au champ de classe n
  }
}
   .....
A a ;
A.f() ;    // appelle la méthode de classe f de la classe A
a.f() ;    // reste autorisé, mais déconseillé
```

7.2.2 Exemple

Voici un exemple illustrant l'emploi d'une méthode de classe. Il s'agit de l'exemple précédent (paragraphe 7.1.2), dans lequel nous avons introduit une méthode de classe nommée *nbObj* affichant simplement le nombre d'objets de sa classe.

```
class Obj
{ public Obj()
  { System.out.print ("++ creation objet Obj ; ") ;
    nb ++ ;
    System.out.println ("il y en a maintenant " + nb) ;
  }
  public static long nbObj ()
  { return nb ;
  }
  private static long nb=0 ;
}
```

```
public class TstObj2
{ public static void main (String args[])
  { Obj a ;
    System.out.println ("Main 1 : nb objets = " + Obj.nbObj() ) ;
    a = new Obj() ;
    System.out.println ("Main 2 : nb objets = " + Obj.nbObj() ) ;
    Obj b ;
    System.out.println ("Main 3 : nb objets = " + Obj.nbObj() ) ;
    b = new Obj() ;
    Obj c = new Obj() ;
    System.out.println ("Main 4 : nb objets = " + Obj.nbObj() ) ;
  }
}
```

```
Main 1 : nb objets = 0
++ creation objet Obj ; il y en a maintenant 1
Main 2 : nb objets = 1
Main 3 : nb objets = 1
++ creation objet Obj ; il y en a maintenant 2
++ creation objet Obj ; il y en a maintenant 3
Main 4 : nb objets = 3
```

Exemple d'utilisation d'une méthode de classe

7.2.3 Autres utilisations des méthodes de classe

En Java, les méthodes de classe s'avèrent pratiques pour permettre à différents objets d'une classe de disposer d'informations collectives. Nous en avons vu un exemple ci-dessus avec le comptage d'objets d'une classe. On pourrait aussi introduire dans une des classes *Point* déjà rencontrées deux champs de classe destinés à contenir les coordonnées d'une origine partagée par tous les points.

Mais les méthodes de classe peuvent également fournir des services n'ayant de signification que pour la classe même. Ce serait par exemple le cas d'une méthode fournissant l'identification d'une classe (nom de classe, numéro d'identification, nom de l'auteur...).

Enfin, on peut utiliser des méthodes de classe pour regrouper au sein d'une classe des fonctionnalités ayant un point commun et n'étant pas liées à un quelconque objet. C'est le cas de la classe *Math* qui contient des fonctions de classe telles que *sqrt*, *sin*, *cos*. Ces méthodes n'ont d'ailleurs qu'un très lointain rapport avec la notion de classe. En fait, ce regroupement est le seul moyen dont on dispose en Java pour retrouver (artificiellement) la notion de fonction indépendante qu'on trouve dans les langages usuels (objet ou non). Notez que c'est cette démarche que nous avons employée pour réaliser la classe *Clavier* qui procure des méthodes (statiques) de lecture au clavier.

7.3 Initialisation des champs de classe

7.3.1 Généralités

Nous avons vu comment les champs usuels se trouvent initialisés : d'abord à une valeur par défaut, ensuite à une valeur fournie (éventuellement) lors de leur déclaration, enfin par le constructeur.

Ces possibilités vont s'appliquer aux champs statiques avec cependant une exception concernant le constructeur. En effet, alors que l'initialisation d'un champ usuel est faite à la création d'un objet de la classe, celle d'un objet statique doit être faite avant la première utilisation de la classe. Cet instant peut bien sûr coïncider avec la création d'un objet, mais il peut aussi la précéder (il peut même n'y avoir aucune création d'objets). C'est pourquoi l'initialisation d'un champ statique se limite à :

- l'initialisation par défaut,
- l'initialisation explicite éventuelle.

Considérez cette classe :

```
class A
{    .....
   public static void f() ;
     .....
   private static int n = 10 ;
   private static int p ;
}
```

Une simple déclaration telle que la suivante entraînera l'initialisation des champs statiques de *A* :

```
A a ;  // aucun objet de type A n'est encore créé, les champs statiques
       // de A sont intialisés : p (implicitement) à 0, n (explicitement) à 10
```

Il en ira de même en cas d'appel d'une méthode statique de cette classe, même si aucun objet n'a encore été créé :

```
A.f() ;    // initialisation des statiques de A, si pas déjà fait
```

Notez cependant qu'un constructeur, comme d'ailleurs toute méthode, peut très bien modifier la valeur d'un champ statique ; mais il ne s'agit plus d'une initialisation, c'est-à-dire d'une opération accompagnant la création du champ.

Enfin, un champ de classe peut être déclaré *final*. Il doit alors obligatoirement recevoir une valeur initiale, au moment de sa déclaration. En effet, comme tout champ déclaré *final*, il ne peut pas être initialisé implicitement. De plus, comme il s'agit d'un champ de classe, il ne peut plus être initialisé par un constructeur.

7.3.2 Bloc d'initialisation statique

Java permet d'introduire dans la définition d'une classe un ou plusieurs blocs d'instructions précédés du mot *static*. Dans ce cas, leurs instructions n'ont accès qu'aux champs statiques de la classe.

Contrairement aux blocs d'initialisation ordinaires (sans *static*) que nous avions déconseillé, les blocs d'initialisation statiques présentent un intérêt lorsque l'initialisation des champs statiques ne peut être faite par une simple expression. En effet, il n'est plus possible de se reposer sur le constructeur, non concerné par l'initialisation des champs statiques. En voici un exemple qui fait appel à la notion de tableau que nous étudierons plus loin[1] :

```
class A
{ private static int t[] ;
  .....
  static { .....
           int nEl = Clavier.lireInt() ;
           t = new int[nEl] ;
           for (int i=0 ; i<nEl ; i++) t[i] = i ;
         }
  .....
}
```

8 Surdéfinition de méthodes

On parle de surdéfinition[2] (ou encore de surcharge) lorsqu'un même symbole possède plusieurs significations différentes entre lesquelles on choisit en fonction du contexte. Sans même en avoir conscience, nous sommes en présence d'un tel mécanisme dans des expressions arithmétiques telles que *a+b* : la signification du symbole + dépend du type des variables *a* et *b*.

En Java, cette possibilité de surdéfinition s'applique aux méthodes d'une classe, y compris aux méthodes statiques. Plusieurs méthodes peuvent porter le même nom, pour peu que le nombre et le type de leurs arguments permettent au compilateur d'effectuer son choix.

8.1 Exemple introductif

Considérons cet exemple, dans lequel nous avons doté la classe *Point* de trois méthodes *deplace* :

- la première à deux arguments de type *int*,
- la deuxième à un seul argument de type *int*,
- la troisième à un seul argument de type *short*.

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

1. Attention, la déclaration de *t* doit précéder son utilisation. Elle doit donc ici être placée avant le bloc d'initialisation statique.

2. En anglais *overload*.

```
  public void deplace (int dx, int dy)  // deplace  (int, int)
  { x += dx ; y += dy ;
  }
  public void deplace (int dx)          // deplace  (int)
  { x += dx ;
  }
  public void deplace (short dx)        // deplace  (short)
  { x += dx ;
  }
  private int x, y ;
}
public class Surdef1
{ public static void main (String args[])
  { Point a = new Point (1, 2) ;
    a.deplace (1, 3) ; // appelle deplace (int, int)
    a.deplace (2) ;    // appelle deplace (int)
    short p = 3 ;
    a.deplace (p) ;    // appelle deplace (short)
    byte b = 2 ;
    a.deplace (b) ;    // appelle deplace (short) apres conversion de b en short
  }
}
```

Exemple de surdéfinition de la méthode deplace *de la classe* Point

Les commentaires en regard des différents appels indiquent quelle est la méthode effectivement appelée. Les choses sont relativement évidentes ici. Notons simplement la conversion de *byte* en *short* dans le dernier appel.

8.2 En cas d'ambiguïté

Supposons que notre classe *Point* précédente ait été dotée (à la place des précédentes) des deux méthodes *deplace* suivantes :

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

Considérons alors ces instructions :

```
Point a = ...
int n ; byte b ;
a.deplace (n, b) ;    // OK : appel de deplace (int, byte)
a.deplace (b, n) ;    // OK : appel de deplace (byte, int)
a.deplace (b, b) ;    // erreur : ambiguïté
```

Le dernier appel sera refusé par le compilateur. Même sans connaître les règles effectivement utilisées dans ce cas, on voit bien qu'il existe deux possibilités apparemment équivalentes :

soit convertir le premier argument en *int* et utiliser *deplace (int, byte*), soit convertir le second argument en *int* et utiliser *deplace (byte, int)*.

En revanche, la présence de conversions implicites dans les évaluations d'expressions arithmétiques peut ici encore avoir des conséquences inattendues :

```
a.deplace (2*b, b) ;   // OK : 2*b de type int --> appel de deplace (int, byte)
```

8.3 Règles générales

À la rencontre d'un appel donné, le compilateur recherche toutes les *méthodes acceptables* et il choisit la meilleure si elle existe. Pour qu'une méthode soit acceptable, il faut :

- qu'elle dispose du nombre d'arguments voulus,
- que le type de chaque argument effectif soit compatible par affectation avec le type de l'argument muet correspondant[1],
- qu'elle soit accessible (par exemple, une méthode privée sera acceptable pour un appel depuis l'intérieur de la classe, alors qu'elle ne le sera pas pour un appel depuis l'extérieur).

Le choix de la méthode se déroule alors ainsi :

- Si aucune méthode n'est acceptable, il y a erreur de compilation.
- Si une seule méthode est acceptable, elle est bien sûr utilisée pour l'appel.
- Si plusieurs méthodes sont acceptables, le compilateur essaie d'en trouver une qui soit *meilleure* que toutes les autres. Pour ce faire, il procède par éliminations successives. Plus précisément, pour chaque paire de méthodes *M* et *M'*, il regarde si tous les arguments (muets, cette fois) de *M* sont compatibles par affectation avec tous les arguments muets de *M'* ; si tel est le cas, *M'* est éliminée (elle est manifestement moins bonne que *M*).

  Après élimination de toutes les méthodes possibles :

  – s'il ne reste plus qu'une seule méthode, elle est utilisée,

  – s'il n'en reste aucune, on obtient une erreur de compilation,

  – s'il en reste plusieurs, on obtient une erreur de compilation mentionnant une ambiguïté.

Remarques

1 Le type de la valeur de retour d'une méthode n'intervient pas dans le choix d'une méthode surdéfinie[2].

---

1. Notez que l'on retrouve les règles habituelles de l'appel d'une méthode non surdéfinie (qui correspond au cas où une seule est acceptable).

2. On notera que si le type d'un argument effectif est parfaitement défini par l'appel d'une méthode, il n'en va plus de même pour la valeur de retour. Au contraire, c'est même la méthode choisie qui définira ce type.

2 On peut surdéfinir des méthodes de classe, de la même manière qu'on surdéfinit des méthodes usuelles.

3 Les arguments déclarés *final* n'ont aucune incidence dans le processus de choix. Ainsi, avec :

```
public void deplace (int dx)       { ..... }
public void deplace (final int dx) { ..... }
```

vous obtiendrez une erreur de compilation (indépendamment de tout appel de *deplace*), comme si vous aviez défini deux foix la même méthode[1]. Bien entendu, ces deux définitions seront acceptées (comme elles le seraient sans *final*) :

```
public void deplace (int dx)       { ..... }
public void deplace (final byte dx) { ..... }
```

4 Les règles de recherche d'une méthode surdéfinie devront être complétées par :

- les possibilités de conversion d'un objet en objet d'une classe de base (étudiées au chapitre 8).
- les possibilités introduites par le JDK 5.0 : conversions entre types primitifs et types enveloppes (étudiées dans le chapitre relatif à l'héritage) ; utilisation éventuelle d'arguments variables en nombre (étudiée dans le chapitre relatif aux tableaux).

C++ En C++

C++ dispose aussi de la surdéfinition des méthodes (et des fonctions ordinaires). Les règles de détermination de la bonne méthode sont toutefois beaucoup plus complexes qu'en Java (l'intuition ne suffit plus toujours !). Contrairement à Java, C++ permet de fixer des valeurs d'arguments par défaut, ce qui peut éviter certaines surdéfinitions.

8.4 Surdéfinition de constructeurs

Les constructeurs peuvent être surdéfinis comme n'importe quelle autre méthode. Voici un exemple dans lequel nous dotons une classe *Point* de constructeurs à 0, 1 ou 2 arguments :

```
class Point
{
  public Point ()                  // constructeur 1 (sans argument)
  { x = 0 ;  y = 0 ;
  }
```

1. Cette règle est liée au mode de transmission des arguments (par valeur). Nous verrons que, dans les deux cas, *deplace* reçoit une copie de l'argument effectif, de sorte que la présence de *final* n'a aucune incidence sur le fonctionnement de la méthode.

```
  public Point (int abs)           // constructeur 2 (un argument)
  { x = y = abs ;
  }
  public Point (int abs, int ord ) // constructeur 3 (deux arguments)
  { x = abs ; y = ord ;
  }

  public void affiche ()
  { System.out.println ("Coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}

public class Surdef2
{ public static void main (String args[])
  { Point a = new Point () ;      // appelle constructeur 1
    a.affiche() ;
    Point b = new Point (5) ;     // appelle constructeur 2
    b.affiche() ;
    Point c = new Point (3, 9) ;  // appelle constructeur 3
    c.affiche() ;
  }
}
```

```
Coordonnees : 0 0
Coordonnees : 5 5
Coordonnees : 3 9
```

Exemple de surdéfinition d'un constructeur

Remarque

Nous verrons plus loin qu'une méthode peut posséder des arguments de type classe. Il est possible de (sur)définir un constructeur de la classe *Point*, de façon qu'il construise un point dont les coordonnées seront identiques à celle d'un autre point fourni en argument. Il suffit de procéder ainsi :

```
public Point (Point a)         // constructeur par copie d'un autre point
  { x = a.x ; y = a.y ;
  }
   .....
Point a = new Point (1, 3) ;    // construction usuelle
Point d = new Point (d) ;       // appel du constructeur par copie d'un point
```

Notez qu'ici la distinction entre copie superficielle et copie profonde n'existe pas (*Point* ne contient aucun champ de type classe). On peut dire que ce constructeur réalise le clonage d'un point.

8.5 Surdéfinition et droits d'accès

Nous avons vu qu'une méthode pouvait être publique ou privée. Dans tous les cas, elle peut être surdéfinie. Cependant, les méthodes privées ne sont pas accessibles en dehors de la classe. Dans ces conditions, suivant son emplacement, un même appel peut conduire à l'appel d'une méthode différente.

```
public class Surdfacc
{ public static void main (String args[])
  { A a = new A() ;
    a.g() ;
    System.out.println ("--- dans main") ;
    int n=2 ; float x=2.5f ;
    a.f(n) ; a.f(x) ;
  }
}
class A
{ public void f(float x)
  { System.out.println ("f(float) x = " + x ) ;
  }
  private void f(int n)
  { System.out.println ("f(int) n = " + n) ;
  }
  public void g()
  { int n=1 ; float x=1.5f ;
    System.out.println ("--- dans g ") ;
    f(n) ; f(x) ;
  }
}
```

```
--- dans g
f(int) n = 1
f(float) x = 1.5
--- dans main
f(float) x = 2.0
f(float) x = 2.5
```

Surdéfinition et droits d'accès

Dans *main*, l'appel *a.f(n)* provoque l'appel de la méthode *f(float)* de la classe *A*, car c'est la seule qui soit acceptable (*f(int)* étant privée). En revanche, pour l'appel comparable *f(n)* effectué au sein de la méthode *g* de la classe *A*, les deux méthodes *f* sont acceptables ; c'est donc *f(int)* qui est utilisée.

9 Échange d'informations avec les méthodes

En Java, la transmission d'un argument à une méthode et celle de son résultat ont toujours lieu par valeur. Comme pour l'affectation, les conséquences en seront totalement différentes, selon que l'on a affaire à une valeur d'un type primitif ou d'un type classe.

9.1 Java transmet toujours les informations par valeur

Dans les différents langages de programmation, on rencontre principalement deux façons d'effectuer le transfert d'information requis par la correspondance entre argument effectif et argument muet :

- *par valeur* : la méthode reçoit une copie de la valeur de l'argument effectif ; elle travaille sur cette copie qu'elle peut modifier à sa guise, sans que cela n'ait d'incidence sur la valeur de l'argument effectif ;
- *par adresse* (ou par référence) : la méthode reçoit l'adresse (ou la référence) de l'argument effectif avec lequel elle travaille alors directement ; elle peut donc, le cas échéant, en modifier la valeur.

Certains langages permettent de choisir entre ces deux modes de transfert. Java emploie systématiquement le premier mode. Mais lorsqu'on manipule une variable de type objet, son nom représente en fait sa référence de sorte que la méthode reçoit bien une copie ; mais il s'agit d'une copie de la référence. La méthode peut donc modifier l'objet concerné qui, quant à lui, n'a pas été recopié. En définitive, tout se passe comme si on avait affaire à une transmission par valeur pour les types primitifs et à une transmission par référence pour les objets.

Les mêmes remarques s'appliquent à la valeur de retour d'une méthode.

9.2 Conséquences pour les types primitifs

Ainsi, une méthode ne peut pas modifier la valeur d'un argument effectif d'un type primitif. Cela est rarement gênant dans un contexte de programmation orientée objet.

Voici cependant un exemple, un peu artificiel, montrant les limites de ce mode de transmission. Supposons qu'on souhaite réaliser une méthode nommée *Échange* permettant d'échanger les valeurs de deux variables de type entier reçues en argument. Comme une telle méthode ne concerne aucun objet, on en fera tout naturellement une méthode de classe[1] d'une classe quelconque, par exemple *Util* (contenant par exemple différentes méthodes utilitaires). Nous pourrions par exemple procéder ainsi (la méthode *main* sert ici à tester notre méthode *Échange* et à prouver qu'elle ne fonctionne pas) :

1. Dans certains langages tels que C++, on utiliserait tout simplement une fonction usuelle. Mais Java oblige à faire de toute fonction une méthode, quitte à ce qu'il s'agisse artificiellement d'une méthode de classe.

```
class Util
{ public static void Échange (int a, int b)  // ne pas oublier static
  { System.out.println ("début Échange : " + a + " " + b) ;
    int c ;
    c = a ; a = b ; b = c ;
    System.out.println ("fin Échange   : " + a + " " + b) ;
  }
}
public class Échange
{ public static void main (String args[])
  { int n = 10, p = 20 ;
    System.out.println ("avant appel   : " + n + " " + p) ;
    Util.Échange (n, p) ;
    System.out.println ("apres appel   : " + n + " " + p) ;
  }
}
```

```
avant appel   : 10 20
debut Échange : 10 20
fin Échange   : 20 10
apres appel   : 10 20
```

Quand la transmission par valeur s'avère gênante

Comme on peut s'y attendre, un échange a bien eu lieu ; mais il a porté sur les valeurs des arguments muets *a* et *b* de la méthode *Échange*. Les valeurs des arguments effectifs *n* et *p* de la méthode *main* n'ont nullement été affectés par l'appel de la méthode *Échange*.

C++ En C++

En C++ on peut traiter le problème précédent en transmettant à une fonction non plus les valeurs de variables, mais leurs adresses, par le biais de pointeurs. Cela n'est pas possible en Java, qui ne dispose pas de pointeurs : c'est d'ailleurs ce qui contribue largement à sa sécurité.

9.3 Cas des objets transmis en argument

Jusqu'ici, les méthodes que nous avons rencontrées ne possédaient que des arguments d'un type primitif. Bien entendu, Java permet d'utiliser des arguments d'un type classe. C'est ce que nous allons examiner ici.

9.3.1 L'unité d'encapsulation est la classe

Supposez que nous voulions, au sein d'une classe *Point*, introduire une méthode nommée *coincide* chargée de détecter la coïncidence éventuelle de deux points. Son appel (par exem-

ple au sein d'une méthode *main*) se présentera obligatoirement sous la forme suivante, *a* étant un objet de type *Point* :

```
a.coincide (...)
```

Il nous faudra donc transmettre le second point en argument ; s'il se nomme *b*, cela nous conduira à un appel de cette forme :

```
a.coincide (b)
```

ou encore, compte tenu de la symétrie du problème :

```
b.coincide (a)
```

Voyons comment écrire la méthode *coincide*. Son en-tête pourrait se présenter ainsi :

```
public boolean coincide (Point pt)
```

Il nous faut comparer les coordonnées de l'objet fourni implicitement lors de l'appel (ses membres étant désignés comme d'habitude par *x* et *y*) avec celles de l'objet *pt* reçu en argument et dont les champs sont alors désignés par *pt.x* et *pt.y*. La méthode *coincide* se présentera ainsi :

```
public boolean coincide (Point pt)
  { return ((pt.x == x) && (pt.y == y)) ;
  }
```

On voit que la méthode *coincide*, appelée pour un objet *a*, est autorisée à accéder aux champs privés d'un autre objet *b* de la même classe. On traduit cela en disant qu'en Java, **l'unité d'encapsulation est la classe et non l'objet**. Notez que nous avions déjà eu l'occasion de signaler que seules les méthodes d'une classe pouvaient accéder aux champs privés de cette classe. Nous voyons clairement ici que cette autorisation concerne bien tous les objets de la classe, et non seulement l'objet courant.

Voici un exemple complet de programme, dans lequel la classe a été réduite au strict minimum :

```
class Point
{ public Point(int abs, int ord)
  { x = abs ; y = ord ;  }
  public boolean coincide (Point pt)
  { return ((pt.x == x) && (pt.y == y)) ;
  }
  private int x, y ;
}
public class Coincide
{ public static void main (String args[])
  { Point a = new Point (1, 3) ;
    Point b = new Point (2, 5) ;
    Point c = new Point (1,3) ;
    System.out.println ("a et b : " + a.coincide(b) + " " + b.coincide(a)) ;
    System.out.println ("a et c : " + a.coincide(c) + " " + c.coincide(a)) ;
  }
}
```

```
a et b : false false
a et c : true true
```

Test de coïncidence de deux points par une méthode

Remarques

1 Bien entendu, lorsqu'une méthode d'une classe *T* reçoit en argument un objet de classe *T'*, différente de *T*, elle n'a pas accès aux champs ou méthodes privées de cet objet.

2 En théorie, le test de coïncidence de deux points est "symétrique" puisque l'ordre dans lequel on considère les deux points est indifférent. Cette symétrie ne se retrouve pas dans la définition de *coincide*, pas plus que dans son appel. Cela provient du mécanisme même d'appel de méthode. On pourrait éventuellement faire effectuer ce test de coïncidence par une méthode de classe, ce qui rétablirait la symétrie, par exemple :

```
class Point
{ public Point(int abs, int ord)
  { x = abs ; y = ord ;
  }
  public static boolean coincide (Point p1, Point p2)
  { return ((p1.x == p2.x) && (p1.y == p2.y)) ;
  }
  private int x, y ;
}
public class Coincid2
{ public static void main (String args[])
  { Point a = new Point (1, 3) ;
    Point b = new Point (2, 5) ;
    Point c = new Point (1,3) ;
    System.out.println ("a et b : " + Point.coincide(a, b) ) ;
    System.out.println ("a et c : " + Point.coincide(a, c) ) ;
  }

}
a et b : false
a et c : true
```

Test de coïncidence de deux points par une méthode statique

9.3.2 Conséquences de la transmission de la référence d'un objet

Comme nous l'avons déjà dit, lors d'un appel de méthode, les arguments sont transmis par recopie de leur valeur. Nous en avons vu les conséquences pour les types primitifs. Dans le cas d'un argument de type objet, en revanche, la méthode reçoit la copie de la référence à l'objet. Elle peut donc tout à fait modifier l'objet correspondant. Cet aspect n'apparaissait pas

dans nos précédents exemples puisque la méthode *coincide* n'avait pas à modifier les coordonnées des points reçus en argument.

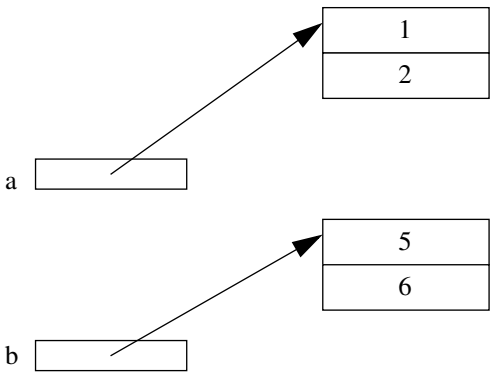
Nous vous proposons maintenant un exemple dans lequel une telle modification est nécessaire. Nous allons introduire dans une classe *Point* une méthode nommée *permute*, chargée d'échanger les coordonnées de deux points. Elle pourrait se présenter ainsi :

```
public void permute (Point a)
{ Point c = new Point(0,0) ;
  c.x = a.x ; c.y = a.y ;  // copie de a dans c
  a.x = x ;   a.y = y ;    // copie du point courant dans a
  x = c.x ;   y = c.y ;    // copie de c dans le point courant
}
```

Cette méthode reçoit en argument la référence *a* d'un point dont elle doit échanger les coordonnées avec celles du point concerné par la méthode. Ici, nous avons créé un objet local *c* de classe *Point* qui nous sert à effectuer l'échange[1]. Illustrons le déroulement de notre méthode. Supposons que l'on ait créé deux points de cette façon :

```
Point a = new Point (1, 2) ;
Point b = new Point (5, 6) ;
```

ce qu'on peut illustrer ainsi :

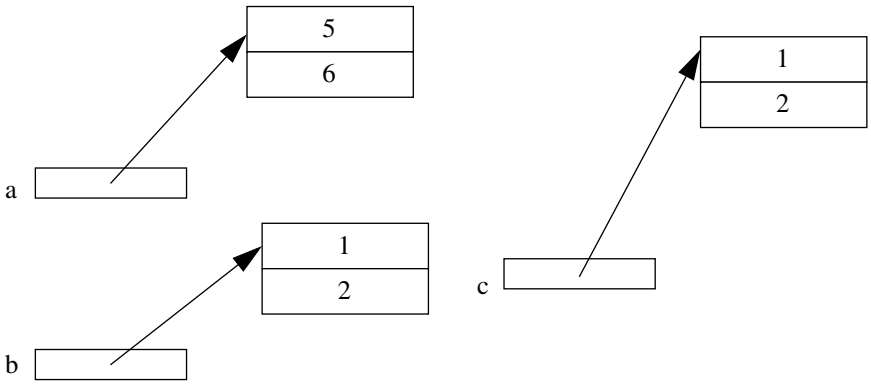


Considérons l'appel

```
a.permute (b) ;
```

1. Nous aurions également pu utiliser deux variables locales de type *int*.

À la fin de l'exécution de la méthode (avant son retour), la situation se présente ainsi :



Notez bien que ce ne sont pas les références contenues dans *a* et *b* qui ont changé, mais seulement les valeurs des objets correspondants. L'objet référencé par *c* deviendra candidat au ramasse-miettes dès la sortie de la méthode *permute*.

Voici un programme complet utilisant cette méthode *permute* :

```
class Point
{
  public Point(int abs, int ord)
  { x = abs ; y = ord ;
  }
  public void permute (Point a)  // methode d'Échange les coordonnees
                                 // du point courant avec celles de a
  { Point c = new Point(0,0) ;
    c.x = a.x ; c.y = a.y ;  // copie de a dans c
    a.x = x ;   a.y = y ;    // copie du point courant dans a
    x = c.x ;   y = c.y ;    // copie de c dans le point courant
  }
  public void affiche ()
  { System.out.println ("Coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
public class Permute
{ public static void main (String args[])
  { Point a = new Point (1, 2) ;
    Point b = new Point (5, 6) ;
    a.affiche() ; b.affiche() ;
    a.permute (b) ;
    a.affiche() ; b.affiche() ;
  }
}
```

```
Coordonnees : 1 2
Coordonnees : 5 6
Coordonnees : 5 6
Coordonnees : 1 2
```

Méthode de permutation des coordonnées de deux points

9.4 Cas de la valeur de retour

Comme on peut s'y attendre, on reçoit toujours la copie de la valeur fournie par une méthode. Là encore, cela ne pose aucun problème lorsque cette valeur est d'un type primitif. Mais une méthode peut aussi renvoyer un objet. Dans ce cas, elle fournit une copie de la référence à l'objet concerné. Voici un exemple exploitant cette remarque où nous dotons une classe *Point* d'une méthode fournissant le symétrique du point concerné.

```
class Point
{
  public Point(int abs, int ord)
  { x = abs ; y = ord ;
  }
  public Point symetrique()
  { Point res ;
    res = new Point (y, x) ;
    return res ;
  }
  public void affiche ()
  { System.out.println ("Coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
public class Sym
{ public static void main (String args[])
  { Point a = new Point (1, 2) ;
    a.affiche() ;
    Point b = a.symetrique() ;
    b.affiche() ;
  }
}
```

```
Coordonnees : 1 2
Coordonnees : 2 1
```

Exemple de méthode fournissant en retour le symétrique d'un point

Notez bien que la variable locale *res* disparaît à la fin de l'exécution de la méthode *symetrique*. En revanche, l'objet créé par *new Point(y, x)* continue d'exister. Comme sa référence est

effectivement copiée par *main* dans *b*, il ne sera pas candidat au ramasse-miettes. Bien entendu, on pourrait envisager la situation suivante :

```
Point p = new Point(2, 5) ;
  .....
for (...)
{ Point s = p.symetrique() ;
  ...
}
```

Si la référence contenue dans *s* n'est pas recopiée dans une autre variable au sein de la boucle *for*, l'objet référencé par *s* (créé par la méthode *symetrique*) deviendra candidat au ramasse-miettes à la fin de la boucle *for.*

9.5 Autoréférence : le mot-clé this

9.5.1 Généralités

Considérons l'application d'une méthode à un objet, par exemple :

```
a.deplace (4, 5) ;
```

Il est évident que cette méthode *deplace* reçoit, au bout du compte, une information lui permettant d'identifier l'objet concerné (ici *a*), afin de pouvoir agir convenablement sur lui.

Bien entendu, la transmission de cette information est prise en charge automatiquement par le compilateur. C'est ce qui permet, dans la méthode (*deplace*), d'accéder aux champs de l'objet sans avoir besoin de préciser sur quel objet on agit.

Mais il peut arriver qu'au sein d'une méthode, on ait besoin de faire référence à l'objet dans sa globalité (et non plus à chacun de ses champs). Ce sera par exemple le cas si l'on souhaite transmettre cet objet en argument d'une autre méthode. Un tel besoin pourrait apparaître dans une méthode destinée à ajouter l'objet concerné à une liste chaînée...

Pour ce faire, Java dispose du mot-clé *this* :

```
class A
{ .....
  public void f(...)   // méthode de la classe A
  { .....  // ici this désigne la référence à l'objet ayant appelé la méthode f
  }
}
```

9.5.2 Exemples d'utilisation de this

À titre d'illustration du rôle de *this*, voici une façon artificielle d'écrire la méthode *coincide* rencontrée au paragraphe 9.3.1 :

```
public boolean coincide (Point pt)
  { return ((pt.x == this.x) && (pt.y == this.y)) ;
  }
```

Notez que l'aspect symétrique du problème apparaît plus clairement.

Ce type de notation artificielle peut s'avérer pratique dans l'écriture de certains constructeurs. Par exemple, le constructeur suivant :

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

peut aussi être écrit ainsi :

```
public Point(int x, int y)  // notez les noms des arguments muets ici
 { this.x = x ;             // ici x désigne le premier argument de Point
                            // le champ x de l'objet courant est masqué ; mais
                            // on peut le nommer this.x
   this.y = x ;
 }
```

Cette démarche permet d'employer des noms d'arguments identiques à des noms de champ, ce qui évite parfois d'avoir à créer de nouveaux identificateurs, comme *abs* et *ord* ici.

9.5.3 Appel d'un constructeur au sein d'un autre constructeur

Nous avons déjà vu qu'il n'était pas possible d'appeler directement un constructeur, comme dans :

```
a.Point(2, 3) ;
```

Il existe cependant une exception : au sein d'un constructeur, il est possible d'en appeler un autre de la même classe (et portant alors sur l'objet courant). Pour cela, on fait appel au mot-clé *this* qu'on utilise cette fois comme un nom de méthode.

Voici un exemple simple d'une classe *Point* dotée d'un constructeur sans argument qui se contente d'appeler un constructeur à deux arguments avec des coordonnées nulles :

```
class Point
{ public Point(int abs, int ord)
  { x = abs ;
    y = ord ;
    System.out.println ("constructeur deux arguments : " + x + " " + y) ;
  }
  public Point()
  { this (0,0) ;  // appel Point (0,0) ; doit etre la premiere instruction
    System.out.println ("constructeur sans argument") ;
  }
  private int x, y ;
}
public class Consthis
{ public static void main (String args[])
  { Point a = new Point (1, 2) ;
    Point b = new Point() ;
  }
}
```

```
constructeur deux arguments : 1 2
constructeur deux arguments : 0 0
constructeur sans argument
```

Exemple d'appel d'un constructeur au sein d'un autre constructeur

D'une manière générale :

> **L'appel *this(...)* doit obligatoirement être la première instruction du constructeur**.

10 La récursivité des méthodes

Java autorise la récursivité des appels de méthodes. Celle-ci peut être :

- directe : une méthode comporte, dans sa définition, au moins un appel à elle-même ;
- croisée : l'appel d'une méthode entraîne l'appel d'une autre méthode qui, à son tour, appelle la méthode initiale (le cycle pouvant d'ailleurs faire intervenir plus de deux méthodes).

On peut appliquer la récursivité aussi bien aux méthodes usuelles qu'aux méthodes de classes (statiques). Voici un exemple classique d'une méthode statique calculant une factorielle de façon récursive :

```
class Util
{ public static long fac (long n)
  { if (n>1) return (fac(n-1) * n) ;
       else return 1 ;
  }
}

public class TstFac
{ public static void main (String [] args)
  { int n ;
    System.out.print ("donnez un entier positif : ") ;
    n = Clavier.lireInt() ;
    System.out.println ("Voici sa factorielle : " + Util.fac(n) ) ;
  }
}
```

```
donnez un entier positif : 8
Voici sa factorielle : 40320
```

Exemple d'utilisation d'une méthode (statique) récursive de calcul de factorielle

Il faut bien voir qu'un appel de la méthode *fac* entraîne une allocation d'espace pour les éventuelles variables locales (ici, il n'y en a aucune), l'argument *n* et la valeur de retour. Or chaque nouvel appel de *fac*, à l'intérieur de *fac*, provoque une telle allocation, sans que les emplacements précédents n'aient été libérés.

Il y a donc une sorte d'empilement des espaces alloués aux informations gérées par la méthode, parallèlement à un empilement des appels de la méthode. Ce n'est que lors de l'exécution de la première instruction *return* que l'on commencera à "dépiler" les appels et les emplacements, donc à libérer de l'espace mémoire.

Voici comment vous pourriez modifier la méthode *fac* pour qu'elle vous permette de suivre ses différents empilements et dépilements :

```
class Util
{ public static long fac (long n)
  { long res ;
    System.out.println ("** entree dans fac : n = " + n) ;
    if (n<=1) res = 1 ;
     else res = fac(n-1) * n ;
    System.out.println ("** sortie de fac :   res = " + res) ;
    return res ;
  }
}
public class TstFac2
{ public static void main (String [] args)
  { int n ;
    System.out.print ("donnez un entier positif : ") ;
    n = Clavier.lireInt() ;
    System.out.println ("Voici sa factorielle : " + Util.fac(n) ) ;
  }
}
```

```
donnez un entier positif : 5
** entree dans fac : n = 5
** entree dans fac : n = 4
** entree dans fac : n = 3
** entree dans fac : n = 2
** entree dans fac : n = 1
** sortie de fac :   res = 1
** sortie de fac :   res = 2
** sortie de fac :   res = 6
** sortie de fac :   res = 24
** sortie de fac :   res = 120
Voici sa factorielle : 120
```

Suivi des empilements et dépilements des appels d'une fonction récursive

Remarque

Nous n'avons programmé la méthode *fac* sous forme récursive que pour l'exemple. Il est clair qu'elle pourrait être écrite de manière itérative classique :

```
public static long fac (long n)
{ long res=1 ;
  for (long i=1 ; i<=n ; i++)
    res *= i ;
  return res ;
}
```

Une méthode récursive est généralement moins efficace (en temps et en espace mémoire) qu'une méthode itérative. Il est conseillé de ne recourir à une démarche récursive que lorsqu'on ne trouve pas de solution itérative évidente.

11 Les objets membres

Comme nous l'avons souligné à plusieurs reprises, les champs d'une classe sont soit d'un type primitif, soit des références à des objets. Dans le second cas, on parle souvent d'*objet membre* pour caractériser cette situation qui peut être facilement mise en œuvre avec ce qui a été présenté auparavant. Nous allons ici commenter un exemple pour mettre l'accent sur certains points qui peuvent s'avérer fondamentaux en conception objet. Cet exemple servira également d'élément de comparaison entre la notion d'objet membre et celle de classe interne présentée un peu plus loin.

Supposons donc que nous disposions d'une classe *Point* classique[1] :

```
class Point
{ public Point(int x, int y)
  { this.x = x ;
    this.y = y ;
  }
  public void affiche()
  { System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;
  }
  private int x, y ;
}
```

Imaginons que nous souhaitions créer une classe *Cercle* permettant de représenter des cercles définis par un centre, objet du type *Point* précédent, et un rayon (flottant). Par souci de simplification, nous supposerons que les fonctionnalités de notre classe *Cercle* se réduisent à :

- l'affichage des caractéristiques d'un cercle (coordonnées du centre et rayon),

1. Si la notation *this.x* ne vous est pas familière, revoyez le paragraphe 9.5.2.

- le déplacement de son centre.

Nous pouvons envisager que notre classe *Cercle* se présente ainsi :

```
class Cercle
{ public Cercle (int x, int y, float r) { ..... } // constructeur
  public void affiche() { ..... }
  public void deplace (int dx, int dy) { ..... }
  private Point c ;    // centre du cercle
  private float r ;    // rayon du cercle
}
```

L'écriture du constructeur ne pose pas de problème ; nous pouvons procéder ainsi[1] :

```
public Cercle (int x, int y, float r)
{ c = new Point (x, y) ;
  this.r = r ;
}
```

En ce qui concerne la méthode *affiche* de la classe *Cercle*, nous pourrions espérer procéder ainsi :

```
 public void affiche()
  { System.out.println ("Je suis un cercle de rayon " + r ) ;
    System.out.print ("  et de centre ") ;
    c.affiche() ;
  }
```

En fait, cette méthode affiche l'information relative à un cercle de la manière suivante (ici, il s'agit d'un cercle de coordonnées 1, 2 et de rayon 5.5) :

```
Je suis un cercle de rayon 5.5
  et de centre Je suis un point de coordonnees 1 2
```

Certes, on trouve bien toute l'information voulue, mais sa présentation laisse à désirer.

Quant à la méthode *deplace*, il n'est pas possible de l'écrire ainsi :

```
void deplace (int dx, int dy)
{ c.x += dx ;   // x n'est pas un champ public de la classe Point ;
                // on ne peut donc pas accéder à c.x
  c.y += dy ;   // idem
}
```

En effet, seules les méthodes d'une classe peuvent accéder aux champs privés d'un objet de cette classe. Or *deplace* est une méthode de *Centre* ; ce n'est pas une méthode de la classe *Point*[2].

Pour pouvoir réaliser la méthode *deplace*, il faudrait que la classe *Point* dispose :

- soit d'une méthode de déplacement d'un point,
- soit de méthodes d'accès et de méthodes d'altération.

---

1. Si la notation *this.r* ne vous est pas familière, revoyez le paragraphe 9.5.2.

2. Notez que s'il en allait autrement, sous prétexte que la classe *Cercle* dispose d'un membre de type *Point*, il suffirait de créer artificiellement des membres dans une méthode pour pouvoir violer le principe d'encapsulation !

Par exemple, si *Point* disposait des méthodes d'accès *getX* et *getY* et des méthodes d'altération *setX* et *setY*, la méthode *deplace* de *Cercle* pourrait s'écrire ainsi :

```
public void deplace (int dx, int dy)
{ c.setX (c.getX() + dx) ;
  c.setY (c.getY() + dy) ;
}
```

Cet exemple montre bien qu'il est difficile de réaliser une bonne conception de classe, c'est-à-dire de définir le bon contrat. Seul un contrat bien spécifié permettra de juger de la possibilité d'utiliser ou non une classe donnée. Bien sûr, l'exemple est ici suffisamment simple pour que la liste de la classe *Point* puisse tenir lieu de contrat, ou encore pour que l'on définisse la classe *Cercle*, sans recourir à la classe *Point*.

À titre indicatif, voici un programme complet utilisant une classe *Cercle* possédant un objet membre de type *Point*, cette dernière étant dotée des fonctions d'accès et d'altération nécessaires :

```
class Point
{ public Point(int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void affiche()
  { System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;
  }
  public int getX() { return x ; }
  public int getY() { return y ; }
  public void setX (int x) { this.x = x ; }
  public void setY (int y) { this.y = y ; }
  private int x, y ;
}
class Cercle
{ public Cercle (int x, int y, float r)
  { c = new Point (x, y) ;
    this.r = r ;
  }
  public void affiche()
  { System.out.println ("Je suis un cercle de rayon " + r) ;
    System.out.println(" et de centre de coordonnees "
                       + c.getX() + " " + c.getY()) ;
  }
  public void deplace (int dx, int dy)
  { c.setX (c.getX() + dx) ;  c.setY (c.getY() + dy) ;
  }
  private Point c ;   // centre du cercle
  private float r ;   // rayon du cercle
}
public class TstCerc
{ public static void main (String args[])
  { Point p = new Point (3, 5) ;  p.affiche() ;
    Cercle c = new Cercle (1, 2, 5.5f) ;  c.affiche();
  }
}
```

```
Je suis un point de coordonnees 3 5
Je suis un cercle de rayon 5.5
 et de centre de coordonnees 1 2
```

Exemple d'une classe Cercle *comportant un objet membre de type* Point

Remarque

La situation d'objet membre correspond à ce ce qu'on nomme généralement la relation *a*[1] (appartenance). Nous verrons que la situation d'héritage correspond à la relation *est*[2].

12 Les classes internes

La notion de classe interne a été introduite par la version 1.1 de Java, essentiellement dans le but de simplifier l'écriture du code de la programmation événementielle. Sa présentation ici se justifie par son lien avec le reste du chapitre et aussi parce que l'on peut utiliser des classes internes en dehors de la programmation événementielle. Mais son étude peut très bien être différée jusqu'au chapitre 12. Et même là, si vous le désirez, vous pourrez vous contenter d'exploiter un schéma de classe anonyme que nous vous présenterons alors (cette notion fondée en partie sur celle de classe interne, utilise en plus l'une des deux notions d'héritage ou d'interface).

12.1 Imbrication de définitions de classe

Une classe est dite interne lorsque sa définition est située à l'intérieur de la définition d'une autre classe. Malgré certaines ressemblances avec la notion d'objet membre étudiée ci-dessus, elle ne doit surtout pas être confondue avec elle, même s'il est possible de l'utiliser dans ce contexte.

La notion de classe interne correspond à cette situation :

```
class E       // définition d'une classe usuelle (dite alors externe)
{ .....       // méthodes et données de la classe E
  class I     // définition d'une classe interne à la classe E
  { .....     // méthodes et données de la classe I
  }
  .....       // autres méthodes et données de la classe E
}
```

1. En anglais *has a*.

2. En anglais *is a*.

Il est très important de savoir que la définition de la classe *I* n'introduit pas d'office de membre de type *I* dans *E*. En fait, la définition de *I* est utilisable au sein de la définition de *E*, pour instancier quand on le souhaite un ou plusieurs objets de ce type. Par exemple, on pourra rencontrer cette situation :

```
class E
{ public void fe()      // méthode de E
  { I i = new I() ;     // création d'un objet de type I ; sa référence est
                        // ici locale à la méthode fe
  }
  class I
  { .....  }
    .....
}
```

Premier schéma d'utilisation de classe interne

On voit qu'ici un objet de classe *E* ne contient aucun membre de type *I*. Simplement, une de ses méthodes (*fe*) utilise le type *I* pour instancier un objet de ce type.

Mais bien entendu, on peut aussi trouver une ou plusieurs références à des objets de type *I* au sein de la classe *E*, comme dans ce schéma :

```
class E
{ .....
  class I
  { .....
  }
  private I i1, i2 ;   // les champs i1 et i2 de E sont des références
                       // à des objets de type I
}
```

Second schéma d'utilisation de classe interne

Ici, un objet de classe *E* contient deux membres de type *I*. Nous n'avons pas précisé comment les objets correspondants seront instanciés (par le constructeur de *E*, par une méthode de *E*...).

12.2 Lien entre objet interne et objet externe

On peut se demander en quoi les situations précédentes diffèrent d'une définition de *I* qui serait externe à celle de *E*. En fait, les objets correspondant à cette situation de classe interne jouissent de trois propriétés particulières.

1. Un objet d'une classe interne est toujours associé, au moment de son instanciation, à un objet d'un classe externe dont on dit qu'il lui a donné naissance. Dans le pre-

mier schéma ci-dessus, l'objet de référence *i* sera associé à l'objet de type *E* auquel sera appliquée la méthode *fe* ; dans le second schéma, rien n'est précisé pour l'instant pour les objets de référence *i1* et *i2*.

2. Un objet d'une classe interne a toujours accès aux champs et méthodes (même privés) de l'objet externe lui ayant donné naissance (attention : ici, il s'agit bien d'un accès restreint à l'objet, et non à tous les objets de cette classe).
3. Un objet de classe externe a toujours accès aux champs et méthodes (même privés) d'un objet d'une classe interne auquel il a donné naissance.

Si le point 1 n'apporte rien de nouveau par rapport à la situation d'objets membres, il n'en va pas de même pour les points 2 et 3, qui permettent d'établir une communication privilégiée entre objet externe et objet interne.

Exemple 1

Voici un premier exemple utilisant le premier schéma du paragraphe 12.2 et illustrant les points 1 et 2 :

```
class E
{ public void fe()
  { I i = new I() ;     // création d'un objet de type I, associé à l'objet
                        // de classe E lui ayant donné naissance (celui qui
                        // aura appelé la méthode fe)
  }
  class I
  { .....
    public void fi()
    { ..... // ici, on a accès au champ ne de l'objet de classe E
            // associé à l'objet courant de classe I
    }
    private int ni ;
  }
  private int ne ;  // champ privé de E
}
   .....
E e1 = new E(), e2 = new E() ;
e1.fe() ;     // l'objet créé par fe sera associé à e1
              // dans fi, ne désignera e1.n
e2.fe() ;     // l'objet créé par fe sera associé à e2
              // dans fi, ne désignera e2.n
```

Exemple 2

Voici un second exemple utilisant le second schéma du paragraphe 12.2 et illustrant les points 1 et 3 :

```
class E
{ public void E()
  { i1 = new I() ; }
  public void fe()
  { i2 = new I() ; }
  public void g ()
  { .....  // ici, on peut accéder non seulement à i1 et i2,
           // mais aussi à i1.ni ou i2.ni
  }
  class I
  { .....
    private int ni ;
  }
  private I i1, i2 ;   // les champs i1 et i2 de E sont des références
                       // à des objets de type I
}
   .....
E e1 = new E() ;  // ici, le constructeur de e1 crée un objet de type I
                  // associé à e1 et place sa référence dans e1.i1 (ici privé)
E e2 = new E() ;  // ici, le constructeur de e2 crée un objet de type I
                  // associé à e1 et place sa référence dans e2.i1 (ici privé)
e1.fe() ;         // la méthode fe crée un objet de type I associé à e1
                  // et place sa référence dans e1.i2
```

Au bout du compte, on a créé ici deux objets de type *I*, associés à *e1* ; il se trouve que (après appel de *fe* seulement), leurs références figurent dans *e1.i1* et *e1.i2*. La situation ressemble à celle d'objets membres (avec cependant des différences de droits d'accès). En revanche, on n'a créé qu'un seul objet de type *I* associé à *e2*.

Remarques

1 Une méthode statique n'est associée à aucun objet. Par conséquent, une méthode statique d'une classe externe ne peut créer aucun objet d'une classe interne.

2 Une classe interne ne peut pas contenir de membres statiques.

12.3 Exemple complet

Au paragraphe 11, nous avons commenté un exemple de classe *Cercle* utilisant un objet membre de type *Point*. Nous vous proposons ici, à simple titre d'exercice, de créer une telle classe en utilisant une classe nommée *Centre*, interne à *Cercle* :

```
class Cercle
{ class Centre    // définition interne a Cercle
  { public Centre (int x, int y)
    { this.x = x ; this.y = y ;
    }
```

```
    public void affiche()
    { System.out.println (x + ", " + y) ;
    }
    private int x, y ;
  }
  public Cercle (int x, int y, double r)
  { c = new Centre (x, y) ;
    this.r = r ;
  }
  public void affiche ()
  { System.out.print ("cercle de rayon " + r + " de centre ") ;
    c.affiche() ;
  }
  public void deplace (int dx, int dy)
  { c.x += dx ; c.y += dy ;   // ici, on a bien acces à x et y
  }
  private Centre c ;
  private double r ;
}

public class TstCercl
{ public static void main (String args[])
  { Cercle c1 = new Cercle(1, 3, 2.5) ;
    c1.affiche() ;
    c1.deplace (4, -2) ;
    c1.affiche() ;
  }
}
```

```
cercle de rayon 2.5 de centre 1, 3
cercle de rayon 2.5 de centre 5, 1
```

Classe Cercle *utilisant une classe interne* Centre

Ici, la classe *Centre* a été dotée d'une méthode *affiche*, réutilisée par la méthode *affiche* de la classe *Cercle*. La situation de classe interne ne se distingue guère de celle d'objet membre. En revanche, bien que la classe *Centre* ne dispose ni de fonctions d'accès et d'altération, ni de méthode *deplace*, la méthode *deplace* de la classe *Cercle* a bien pu accéder aux champs privés *x* et *y* de l'objet de type *Centre* associé.

Informations complémentaires

Nous venons de vous présenter l'essentiel des propriétés des classes internes. Voici quelques compléments concernant des possibilités rarement exploitées.

Déclaration et instanciation d'un objet d'une classe interne

Nous avons vu comment déclarer et instancier un objet d'une classe interne depuis une classe englobante, ce qui constitue la démarche la plus naturelle. En théorie, Java permet d'utiliser une classe interne depuis une classe indépendante (non englobante). **Mais, il faut quand même rattacher un objet d'une classe interne à un objet de sa classe englobante, moyennant l'utilisation d'une syntaxe particulière de *new***. Supposons que l'on ait :

```
public class E      // classe englobante de I
{ .....
  public class I    // classe interne à E
  { .....
  }
  .....
}
```

En dehors de *E*, vous pouvez toujours déclarer une référence à un objet de type *I*, de cette manière :

```
E.I i ;     // référence à un objet de type I (interne à E)
```

Mais la création d'un objet de type *I* ne peut se faire qu'en le rattachant à un objet de sa classe englobante. Par exemple, si l'on dispose d'un objet *e* créé ainsi :

```
E e = new E() ;
```

on pourra affecter à *i* la référence à un objet de type *I*, rattaché à *e*, en utilisant *new* comme suit :

```
i = new e.I() ;   // création d'un objet de type I, rattaché à l'objet e
                  // et affectation de sa référence à i
```

Classes internes locales

Vous pouvez définir une classe interne *I* dans une méthode *f* d'une classe *E*. Dans ce cas, l'instanciation d'objets de type *I* ne peut se faire que dans *f*. En plus des accès déjà décrits, un objet de type *I* a alors accès aux variables locales finales de *f*.

```
public class E
{ .....
  void f()
  { final int n=15 ; float x ;
    class I    // classe interne à E, locale à f
    { .....    // ici, on a accès à n, pas à x
    }
    I i = new I() ;   // classique
  }
}
```

Classes internes statiques

Les objets des classes internes dont nous avons parlé jusqu'ici étaient toujours associés à un objet d'une classe englobante. On peut créer des classes internes "autonomes" en employant l'attribut *static* :

```
public class E              // classe englobante
{ .....
  public static class I     // définition (englobée dans celle de E)
  {  .....                  // d'une classe interne autonome
  }
}
```

Depuis l'extérieur de *E*, on peut instancier un objet de classe *I* de cette façon :

```
E.I i = new E.I() ;
```

L'objet *i* n'est associé à aucun objet de type *E*. Bien entendu, la classe *I* n'a plus accès aux membres de *E*, sauf s'il s'agit de membres statiques.

13 Les paquetages

La notion de paquetage correspond à un regroupement logique sous un identificateur commun d'un ensemble de classes. Elle est proche de la notion de bibliothèque que l'on rencontre dans d'autres langages. Elle facilite le développement et la cohabitation de logiciels conséquents en permettant de répartir les classes correspondantes dans différents paquetages. Le risque de créer deux classes de même nom se trouve alors limité aux seules classes d'un même paquetage.

13.1 Attribution d'une classe à un paquetage

Un paquetage est caractérisé par un nom qui est soit un simple identificateur, soit une suite d'identificateurs séparés par des points, comme dans :

MesClasses

Utilitaires.Mathematiques

Utilitaires.Tris

L'attribution d'un nom de paquetage se fait au niveau du fichier source ; toutes les classes d'un même fichier source appartiendront donc toujours à un même paquetage. Pour ce faire, on place, en début de fichier, une instruction de la forme :

```
package xxxxxx ;
```

dans laquelle *xxxxxx* représente le nom du paquetage.

Cette instruction est suffisante, même lorsque le fichier concerné est le premier auquel on attribue le nom de paquetage en question. En effet, la notion de paquetage est une notion "logique", n'ayant qu'un rapport partiel avec la localisation effective des classes ou des fichiers au sein de répertoires[1].

1. Certains environnements peuvent cependant imposer des contraintes quant aux noms de répertoires et à leur localisation.

De même, lorsqu'on recourt à des noms de paquetages hiérarchisés (comme *Utilitaires.Tris*), il ne s'agit toujours que d'une facilité d'organisation logique des noms de paquetage. En effet, on ne pourra jamais désigner simultanément deux paquetages tels que *Utilitaires.Mathematiques* et *Utilitaires.Tris* en se contentant de citer *Utilitaires*. Qui plus est, ce dernier pourra très bien correspondre à d'autres classes, sans rapport avec les précédentes.

En l'absence d'instruction *package* dans un fichier source, le compilateur considère que les classes correspondantes appartiennent au paquetage par défaut. Bien entendu, celui-ci est unique pour une implémentation donnée.

13.2 Utilisation d'une classe d'un paquetage

Lorsque, dans un programme, vous faites référence à une classe, le compilateur la recherche dans le paquetage par défaut. Pour utiliser une classe appartenant à un autre paquetage, il est nécessaire de fournir l'information correspondante au compilateur. Pour ce faire, vous pouvez :

- citer le nom du paquetage avec le nom de la classe,
- utiliser une instruction *import* en y citant soit une classe particulière d'un paquetage, soit tout un paquetage.

En citant le nom de la classe

Si vous avez attribué à la classe *Point* le nom de paquetage *MesClasses* par exemple, vous pourrez l'utiliser simplement en la nommant *MesClasses.Point*. Par exemple :

```
MesClasses.Point p = new MesClasses.Point (2, 5) ;
   .....
p.affiche() ;    // ici, le nom de paquetage n'est pas requis
```

Evidemment, cette démarche devient fastidieuse dès que de nombreuses classes sont concernées.

En important une classe

L'instruction *import* vous permet de citer le nom (complet) d'une ou plusieurs classes, par exemple :

```
import MesClasses.Point, MesClasses.Cercle ;
```

À partir de là, vous pourrez utiliser les classes *Point* et *Cercle* sans avoir à mentionner leur nom de paquetage, comme si elles appartenaient au paquetage par défaut.

En important un paquetage

La démarche précédente s'avère elle aussi fastidieuse dès qu'un certain nombre de classes d'un même paquetage sont concernées. Avec :

```
import MesClasses.* ;
```

vous pourrez ensuite utiliser toutes les classes du paquetage *MesClasses* en omettant le nom de paquetage correspondant.

Précautions

L'instruction :

```
import MesClasses ;
```

ne concerne que les classes du paquetage *MesClasses*. Si, par exemple, vous avez créé un paquetage nommé *MesClasses.Projet1*, ses classes ne seront nullement concernées.

Remarques

1 En citant tout un paquetage dont certaines classes sont inutilisées, vous ne vous pénaliserez ni en temps de compilation, ni en taille des *byte codes*, Bien entendu, si vous devez créer deux paquetages contenant des classes de même nom, cette démarche ne sera plus utilisable (importer deux classes de même nom constitue une erreur).

2 La plupart des environnements imposent des contraintes quant à la localisation des fichiers correspondant à un paquetage (il peut s'agir de fichiers séparés, mais aussi d'archives JAR ou ZIP). En particulier, un paquetage de nom *X.Y.Z* se trouvera toujours intégralement dans un sous-répertoire de nom *X.Y.Z* (les niveaux supérieurs étant quelconques). En revanche, le paquetage *X.Y.U* pourra se trouver dans un sous-répertoire *X.Y.U* rattaché à un répertoire différent du précédent. Avec le SDK[1] de SUN, la recherche d'un paquetage (y compris celle du paquetage courant) se fait dans les répertoires déclarés dans la variable d'environnement *CLASSPATH* (le point y désigne le repértoire courant).

13.3 Les paquetages standard

Les nombreuses classes standard avec lesquelles Java est fourni sont structurées en paquetages. Nous aurons l'occasion d'utiliser certains d'entre eux par la suite, par exemple *java.awt*, *java.awt.event*, *javax.swing*...

Par ailleurs, il existe un paquetage particulier nommé *java.lang* qui est automatiquement importé par le compilateur. C'est ce qui vous permet d'utiliser des classes standard telles que *Math*, *System*, *Float* ou *Integer*, sans avoir à introduire d'instruction *import*.

1. Nouveau nom du JDK depuis Java 1.3.

13.4 Paquetages et droits d'accès

13.4.1 Droits d'accès aux classes

Pour vous permettre de commencer à écrire de petits programmes, nous vous avons déjà signalé qu'un fichier source pouvait contenir plusieurs classes, mais qu'une seule pouvait avoir l'attribut *public*. C'est d'ailleurs ainsi que nous avons procédé dans bon nombre d'exemples.

D'une manière générale, chaque classe dispose de ce qu'on nomme un droit d'accès (on dit aussi un modificateur d'accès). Il permet de décider quelles sont les autres classes qui peuvent l'utiliser. Il est simplement défini par la présence ou l'absence du mot-clé *public* :

- avec le mot-clé *public*, la classe est accessible à toutes les autres classes (moyennant éventuellement le recours à une instruction *import*) ;
- sans le mot-clé *public,* la classe n'est accessible qu'aux classes du même paquetage.

Tant que l'on travaille avec le paquetage par défaut, l'absence du mot *public* n'a guère d'importance (il faut toutefois que la classe contenant *main* soit publique pour que la machine virtuelle y ait accès).

13.4.2 Droits d'accès aux membres d'une classe

Nous avons déjà vu qu'on pouvait utiliser pour un membre (champ ou méthode) l'un des attributs *public* ou *private*. Avec *public*, le membre est accessible depuis l'extérieur de la classe ; avec *private*, il n'est accessible qu'aux méthodes de la classe. En fait, il existe une troisième possibilité, à savoir l'absence de mot-clé (*private* ou *public*). Dans ce cas, l'accès au membre est limité aux classes du même paquetage (on parle d'accès de paquetage). Voyez cet exemple :

```
package P1 ;
public class A  // accessible partout
{ .....
  void f1() { ..... }
  public void f2() { ..... }
}
```

```
package P2 ;
class B   // accessible que de P2
{ .....
  public void g()
  { A a ;
    a.f1() ;   // interdit
    a.f2() ;   // OK
}
```

Remarques

1 Ne confondez pas le droit d'accès à une classe avec le droit d'accès à un membre d'une classe, même si certains des mots-clés utilisés sont communs. Ainsi, *private* a un sens pour un membre, il n'en a pas pour une classe.

2 Nous verrons au chapitre consacré à l'héritage qu'il existe un quatrième droit d'accès aux membres d'une classe, à savoir *protected* (protégé).

Informations complémentaires

Les classes internes sont concernées par ce droit d'accès, mais sa signification est différente[1], compte tenu de l'imbrication de leur définition dans celle d'une autre classe :

- avec *public*, la classe interne est accessible partout où sa classe externe l'est ;
- avec *private* (qui est utilisable avec une classe interne, alors qu'il ne l'est pas pour une classe externe), la classe interne n'est accessible que depuis sa classe externe ;
- sans aucun mot-clé, la classe interne n'est accessible que depuis les classes du même paquetage.

D'une manière générale, l'annexe A récapitule le rôle de ces différents droits d'accès pour les différentes entités que sont les classes, les classes internes, les membres et les interfaces.

1. Elle s'apparente à celle qui régit les droits d'accès à des membres.