9

Classes et objets

Dans le premier chapitre, nous avons fait une distinction entre :

- Les langages procéduraux, disposant de la notion de fonction, outil qui permet de structurer un programme en le décomposant en des parties relativement indépendantes.
- Les langages objet, disposant en plus des notions de classe et d'objet ; comme nous le verrons, les classes permettent également de structurer un programme en le décomposant en des parties autonomes.

Dans ce chapitre, nous allons introduire ces notions de classes et d'objets. Nous serons amenés à distinguer la définition des classes, la création des objets et leur utilisation. Nous verrons ensuite ce qu'est précisément ce que l'on nomme l'encapsulation des données et quelles en sont les conséquences. Puis, nous introduirons l'importante notion de constructeur. Enfin, nous donnerons quelques éléments concernant les deux modes possibles de gestion des objets, à savoir par référence ou par valeur.

1 Introduction

Le concept d'objet consiste à regrouper dans une même entité des données qu'on nomme des attributs (ou encore des champs) et des fonctions qu'on nomme méthodes (ou, parfois, fonctions membres). Seules les méthodes sont habilitées à manipuler ces données, qu'il s'agisse de les modifier ou plus simplement d'en utiliser la valeur. On traduit souvent cette propriété en disant que les données sont encapsulées dans l'objet, autrement dit qu'elles ne sont plus visibles « de l'extérieur » de l'objet. La seule façon d'exploiter les possibilités offertes par l'objet sera de faire appel à ses méthodes.

La notion de classe généralise aux objets la notion de type : une classe n'est rien d'autre qu'une description (unique) pouvant donner naissance à différents objets disposant de la même structure de données (mêmes noms et types d'attributs) et des mêmes méthodes. Différents objets d'une même classe se distinguent par les valeurs de leurs attributs ; en revanche, ils partagent les mêmes méthodes. On trouvait une situation comparable avec deux variables d'un type `réel` qui pouvaient différer par leur valeur ; leur structure de données (réduite ici à une valeur de type `réel`) et leurs « fonctionnalités » (addition, soustraction...) étaient communes.

Généralement, en programmation orientée objet, soit on définit une classe que l'on pourra utiliser ensuite pour créer un ou plusieurs objets de cette classe, soit on utilise des classes existantes (fournies avec le langage ou créées par vous-même ou par d'autres programmeurs). On retrouve là encore quelque chose de comparable à ce qui se passait avec les fonctions.

2 Un premier exemple : une classe Point

Pour introduire ces nouvelles possibilités de classe, objet et encapsulation, nous vous proposons de voir à la fois comment définir et utiliser une nouvelle classe nommée `Point`, permettant de manipuler des points d'un plan. Nous souhaitons qu'elle dispose des trois méthodes suivantes :

- `initialise` pour attribuer des valeurs aux coordonnées d'un point (ici, nous utiliserons le système le plus courant de « coordonnées cartésiennes ») ;
- `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.

En ce qui concerne ses attributs, nous choisirons d'utiliser deux entiers[1] représentant les coordonnées d'un point ; déjà à ce niveau, signalons que rien ne nous empêcherait d'utiliser d'autres informations (coordonnées polaires, par exemple), dans la mesure où ces informations ne seront pas exploitées directement à l'extérieur de la classe.

Dans un premier temps, nous supposerons que notre classe a été convenablement définie et nous allons voir comment l'exploiter pour donner naissance à des objets représentant des points et comment les manipuler. Il nous sera ensuite plus facile de revenir sur la « définition » de la classe elle-même.

1. En toute rigueur, suivant l'usage que l'on sera amené à faire de ces «points », le type réel pourra parfois s'avérer plus adapté.

2.1 Utilisation de notre classe Point

2.1.1 Le mécanisme déclaration, instanciation

A priori, nous pourrions supposer que, de même qu'une déclaration telle que :

```
entier n
```

réserve un emplacement pour une variable de type `entier`,

une déclaration telle que :

```
Point p
```

réserve un emplacement pour un « objet de type `Point` », c'est-à-dire en fait un emplacement pour chacun de ses attributs (les méthodes, communes à tous les objets du type `Point`, n'ayant, quant à elles pas besoin d'être dupliquées pour chaque objet).

Cependant, pour des questions de souplesse d'exploitation des possibilités de programmation orientée objet, tous les langages objet permettent de fonctionner en deux étapes :

- D'une part, la déclaration précédente :

  ```
  Point p
  ```

  réserve simplement un emplacement pour une variable nommée `p`, destinée à recevoir la référence (adresse) d'un objet de type `Point`. Pour l'instant, la valeur de cette variable n'est pas encore définie.

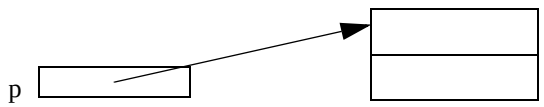
- D'autre part, il existe un mécanisme permettant de réserver l'emplacement mémoire pour un objet, analogue à ce que nous avons appelé « gestion dynamique » dans le cas des tableaux (paragraphe 11 du chapitre 7, page 128). Dans le cas des objets, on parle d'instanciation ou de création pour désigner ce mécanisme). Ici, nous conviendrons que cette instanciation est réalisée par l'expression suivante :

  ```
  Création Point
  ```

  En affectant cette valeur, par exemple à la variable `p` précédente :

  ```
  p := Création Point
  ```

nous aboutissons à une situation que l'on peut schématiser ainsi (les deux cases vides représentant les attributs d'un objet de type `Point`) :



On notera bien qu'une telle démarche fait intervenir :

- Une déclaration classique (ici d'une référence à un objet de type `Point`) ; elle sera exploitée lors de la traduction du programme pour réserver l'emplacement mémoire correspondant.
- Une instruction d'instanciation qui donnera effectivement naissance à l'objet en réservant son emplacement uniquement au moment de l'exécution de l'instruction correspondante.

Comme nous le verrons par la suite, la valeur de `p` pourra très bien évoluer pendant l'exécution, par le biais d'instructions d'affectation.

Remarques

1 Notez bien le vocabulaire que nous avons convenu d'utiliser : nous disons que `p` est une variable de type `Point`, tandis que l'objet référencé par `p` est un objet de type `Point`. En revanche, il nous arrivera souvent de commettre l'abus de langage consistant à parler d'un point pour un « objet de type `Point` », voire même parfois de l'objet `p` au lieu de l'objet référencé par `p`.

   En ce qui concerne le terme instanciation, nous l'appliquerons indifféremment à une classe (il désignera la création d'un objet de cette classe) ou à un objet (il désignera la création de cet objet).

2 Certains langages disposent, en plus de ce mécanisme d'instanciation à l'exécution, de la possibilité de réserver l'emplacement d'un objet par une déclaration. Nous reviendrons plus loin sur cette « dualité » concernant la gestion des objets.

2.1.2 Utilisation d'objets de type Point

Supposons donc que notre variable `p` contienne la référence d'un objet de type `Point` et voyons maintenant comment utiliser cet objet.

Tout d'abord, rappelons que les attributs de nos objets sont encapsulés dans l'objet et qu'il n'est pas possible d'y accéder directement ; d'ailleurs, pour l'instant, nous ne savons même pas comment ils se nomment ! Nous devons obligatoirement utiliser les méthodes de la classe `Point`. Ici, nous avons prévu que la méthode `initialise` fournisse des valeurs aux coordonnées d'un point. Pour l'appeler, nous devrons naturellement préciser :

- les valeurs des paramètres requis : ici, deux entiers représentant les deux coordonnées cartésiennes, par exemple 5 et 8 ;
- l'objet auquel la méthode doit s'appliquer : ici celui référencé par `p`.

Nous utiliserons pour cela une notation très répandue :

```
p.initialise (5, 8) // appelle la méthode initialise de l'objet référencé par p
```

Remarque

On dit parfois que l'instruction :

```
p.initialise (5, 8)
```

« envoie le message » `initialise (5, 8)` à l'objet `p`.

2.2 Définition de la classe Point

Jusqu'ici, nous avions supposé que la classe `Point` existait déjà. Voyons maintenant comment la définir.

Nous conviendrons que la définition d'une classe se fait suivant ce canevas :

```
classe Point
{  // déclaration des attributs
   // définition des méthodes
}
```

En ce qui concerne les attributs, nous choisirons d'utiliser deux entiers représentant les coordonnées cartésiennes d'un point (mais nous verrons que d'autres choix seraient possibles, indépendamment des coordonnées utilisées dans les méthodes). Nous les déclarerons de façon classique, comme on le ferait pour les variables d'un programme ou les variables locales à une fonction :

```
entier abs    // abscisse d'un point
entier ord    // ordonnée d'un point
```

ou, encore :

```
entier abs, ord    // abscisse et ordonnée d'un point
```

Quant à la définition des méthodes, elle se composera, comme celle d'une fonction d'un en-tête en précisant les paramètres (nom de paramètre « muet » et type) ainsi que le type du résultat (ici, aucune de nos méthodes n'en fournit). Si l'on considère par exemple la méthode `initialise` dont on a vu qu'elle serait appelée par une instruction de la forme :

```
p.initialise (5, 8)
```

on constate qu'elle devra disposer de deux paramètres correspondant aux coordonnées à attribuer au point concerné. Nous conviendrons d'écrire son en-tête de cette façon (en utilisant le mot `méthode` à la place du mot `fonction`) :

```
méthode initialise (entier x, entier y)
```

Dans le corps de cette méthode, nous allons devoir affecter la valeur du paramètre muet `x` à l'attribut `abs` de l'objet ayant appelé la méthode. Nous nous contenterons d'écrire cela tout simplement de la manière suivante, sans préciser de quel objet il s'agit :

```
abs := x  // affecte la valeur de x à l'attribut abs de l'objet concerné
```

En effet, nous conviendrons que, dans une méthode, un nom d'attribut désigne l'attribut correspondant de l'objet ayant effectué l'appel. Notez qu'un tel objet est parfaitement connu lors de l'appel ; nous convenons seulement que l'information corrrespondante sera bien transmise à la méthode par des instructions appropriées mises en place par le traducteur du programme (en toute rigueur, tout se passe comme si une méthode disposait d'un paramètre supplémentaire représentant l'objet l'ayant appelé).

2.3 En définitive

Finalement, voici la définition complète de notre classe *Point* :

```
classe Point
{ methode initialise (entier x, entier y)
  { abs := x
    ord := y
  }
  méthode deplace (entier dx, entier dy)
  { abs := abs + dx
    ord := ord + dy
  }
  méthode affiche
  { écrire «Je suis un point de coordonnées », abs, « », ord
  }
  entier abs   // abscisse
  entier ord   // ordonnée
}
```

Définition d'une classe Point

Voici un petit programme utilisant cette classe `Point` :

```
Point p, r              // deux variables de type Point
p := Création Point     // création d'un objet de type Point
p.initialise(3, 5)      // appel de la méthode initialise sur l'objet référencé par p
p.affiche()             // appel de la méthode affiche sur l'objet référencé par p
p.deplace(2, 0)
p.affiche()
r := Création Point     // création d'un autre objet de type Point
r.initialise (6, 8)     // appel de la méhtode initialise sur l'objet référencé par r
r.affiche()
```

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

Utilisation de la classe Point

2.4 Indépendance entre classe et programme

Pour l'instant, nous avons considéré séparément la classe et le programme l'utilisant, sans nous préoccuper de la manière dont ces deux éléments allaient interagir.

En pratique, il va falloir faire en sorte que le programme et la classe soient convenablement réunis pour l'exécution du programme. La démarche utilisée dépendra étroitement du langage et de l'environnement concernés, ainsi que du mode de traduction (compilation, interprétation, code intermédiaire). Dans tous les cas, comme on peut l'espérer, plusieurs

programmes pourront utiliser une même classe qu'il suffira d'avoir écrite et mise au point une fois pour toutes. On retrouve simplement ici une généralisation de ce que nous avions indiqué à propos des fonctions.

Exercice 9.1 Ajouter à la définition de la classe `Point` précédente, une méthode `premQuad` fournissant la valeur `vrai` si le point concerné appartient au « premier quadrant », c'est-à-dire si ses coordonnées sont toutes deux positives ou nulles, et la valeur `faux` dans le cas contraire. Écrire un petit programme utilisant cette nouvelle classe `Point`.

3 L'encapsulation et ses conséquences

3.1 Méthodes d'accès et d'altération

Nous avons vu que les attributs sont encapsulés dans l'objet et qu'il n'est pas possible d'y accéder en dehors des méthodes elles-mêmes. Ainsi, avec notre classe `Point` précédente, nous ne pouvons pas connaître ou modifier l'abscisse d'un point en nous référant directement à l'attribut `abs` correspondant :

```
Point p
p := Création Point
   .....
écrire p.abs      // interdit
p.abs := 9        // interdit
```

Cette contrainte pourra sembler draconienne dans certains cas. Mais, il faut bien comprendre qu'il reste toujours possible de doter une classe de méthodes appropriées permettant :

- d'obtenir la valeur d'un attribut donné ; nous parlerons de « méthodes d'accès » ;
- de modifier la valeur d'un ou de plusieurs attributs ; nous parlerons de « méthodes d'altération »[1].

Par exemple, en plus de la méthode initialise, nous pourrions doter notre classe Point de deux méthodes d'altération `fixeAbs` et `fixeOrd` :

```
méthode fixeAbs (entier x)
{ abs := x
}
méthode fixeOrd (entier y)
{ ord := y
}
```

De même, nous pourrions la doter de deux méthodes d'accès `valeurAbs` et `valeurOrd` :

```
entier méthode valeurAbs
```

1. La dénomination de ces méthodes est loin d'être universelle. Parfois, on rencontre le terme « méthode d'accès » pour qualifier les deux types de méthodes.

```
{ retourne abs
}
entier méthode valeurOrd
{ retourne ord
}
```

On peut alors légitimement se poser la question de l'intérêt d'encapsuler les attributs si on peut les connaître ou les modifier à volonté à l'aide de méthodes d'accès ou d'altération. Pourquoi ne pas en autoriser l'accès direct ? En fait, la réponse réside dans la distinction entre ce que l'on nomme l'interface d'une classe et son implémentation et dont nous allons parler maintenant.

3.2 Notions d'interface, de contrat et d'implémentation

L'interface d'une classe corresond aux informations dont on doit pouvoir disposer pour pouvoir l'utiliser. Il s'agit :

- du nom de la classe ;
- de la signature (nom de la fonction et type des paramètres) et du type du résultat éventuel de chacune de ses méthodes.

Dans le cas de notre classe `Point`, ces informations pourraient se résumer ainsi :

```
classe Point
{ méthode initialise (entier, entier)
  méthode déplace (entier, entier)
  méthode affiche
}
```

Le contrat d'une classe correspond à son interface et à la définition du rôle de ses méthodes.

Enfin, l'implémentation d'une classe correspond à l'ensemble des instructions de la classe, écrites en vue de réaliser le contrat voulu.

L'un des éléments majeurs de la programmation orientée objet est qu'une classe peut tout à fait modifier son implémentation, sans que ceci n'ait de conséquences sur son utilisation (à condition, bien sûr de respecter le contrat !). Par exemple, nous pourrions très bien décider que nos points seront représentés, non plus par leurs coordonnées cartésiennes, mais par leurs coordonnées « polaires »[1]. Bien entendu, il faudrait adapter en conséquences le corps des méthodes, lesquelles devraient conserver leur signature et leur signification : autrement dit, `initialise` devrait continuer à recevoir des coordonnées cartésiennes. Il va de soi qu'ici, une telle modification paraîtrait fortement fantaisiste puisqu'elle entraînerait des complications inutiles. Mais, imaginez une classe `Point` plus riche dotée de méthodes travaillant, les unes en coordonnées cartésiennes, les autres en coordonnées polaires. Dans ces conditions, il faudrait bien sûr trancher dans le choix des attributs entre coordonnées cartésiennes ou coordonnées polaires. Mais ceci n'aura pas de répercussion sur l'interface de la classe. Si celle-ci

1. En coordonnées polaires, un point est caractérisé par sa distance à l'origine (rayon vecteur) et l'angle que fait le segment joignant l'origine au point avec l'axe des abscisses.

est dotée de méthodes telles que `fixeAbs`, on ne saura pas si celle-ci fixe la valeur d'un attribut (`abs`) ou si elle modifie en conséquences les coordonnées polaires.

Exercice 9.2 Écrire une classe `Point` ne disposant que des 4 méthodes `fixeAbs`, `fixeOrd`, `valeurAbs` et `valeurOrd`. Réécrire le programme du paragraphe 2.3, page 184, en utilisant cette nouvelle classe.

3.3 Dérogations au principe d'encapsulation

Nous avons indiqué que, en programmation orientée objet, les attributs étaient encapsulés dans la classe et nous avions donc supposé qu'il en allait ainsi des attributs `abs` et `ord` de notre classe `Point`.

Cependant, la plupart des langages offrent une certaine latitude sur ce point, en autorisant que certains attributs restent accessibles à un programme utilisant la classe. On est alors amené à parler du statut d'accès (ou plus simplement de l'accès) d'un attribut qui peut alors être :

- privé : c'est le cas usuel que nous avons considéré jusqu'ici ;
- public : dans ce cas, l'attribut est directement lisible ou modifiable.

Si nous souhaitions déclarer qu'un attribut tel que `abs` est public, nous procéderions ainsi

```
public entier abs  // l'attribut abs n'est plus encapsulé
```

Dans ce cas, ayant instancié un objet de type `Point`, référencé par exemple par la variable `p`, nous pourrions utiliser directement cet attribut :

```
écrire p.abs
```

ou, pire :

```
p.abs := 15
```

Mais, nous n'exploiterons de telles possibilités que de manière exceptionnelle, éventuellement à titre de contre-exemple.

Par ailleurs, il est généralement possible de prévoir un accès privé à une méthode, de sorte qu'elle ne soit plus accessible en dehors de la classe elle-même. Une telle démarche peut s'avérer intéressante lorsqu'il s'agit d'introduire dans l'implémentation de la classe une « méthode de service », utilisée par exemple par plusieurs autres méthodes, mais n'ayant rien à voir avec le contrat de la classe puisque non prévue dans son interface.

D'une manière générale, nous conviendrons que :

> Par défaut, les attributs d'une classe sont privés et les méthodes sont publiques.

On peut modifier explicitement ce statut en ajoutant le mot `public` dans la déclaration de l'attribut ou le mot `privé` dans l'en-tête de la méthode.

Remarques

1 On notera que le statut d'un attribut porte en bloc sur l'accès ou l'altération de cet attribut. Peu de langages permettent de différencier les deux actions, en les dotant d'autorisations différentes.

2 Nous verrons qu'il existe d'autres statuts d'accès, liés à la notion d'héritage.

3 On rencontre parfois le terme « rétention d'information », à la place d'encapsulation.

4 Méthode appelant une autre méthode

Jusqu'ici, nous avons appelé une méthode d'une classe en l'appliquant à un objet. Mais, de même qu'une fonction pouvait en appeler une autre, une méthode peut en appeler une autre.

Pour l'instant, nous n'envisagerons pas le cas où une méthode d'une classe appelle une méthode d'une autre classe, car il fait souvent intervenir la notion de composition d'objets dont nous parlerons plus tard. En revanche, nous pouvons considérer le cas où une méthode d'une classe appelle une autre méthode de la même classe. Considérons cette situation :

```
classe Point
{ méthode afficheAbs { ..... }   // affiche l'abscisse
  méthode afficheOrd { ..... }   // affiche l'ordonnée
  méthode affiche    { ..... }   // affiche l'abscisse et l'ordonnée
}
```

La méthode `affiche` peut chercher à utiliser les méthodes `afficheAbs` et `afficheOrd`. Cela est tout à fait possible en procédant ainsi :

```
méthode affiche
 { afficheAbs
   afficheOrd
 }
```

On notera bien que, là encore, il n'est pas besoin de préciser l'objet auxquels s'appliquent les appels `afficheAbs` et `afficheOrd`. Il s'agit par convention de l'objet (unique à un moment donné) ayant appelé `affiche`.

5 Les constructeurs

5.1 Introduction

Considérons à nouveau notre classe `Point` du paragraphe 2.3, page 184, dotée de ses méthodes `initialise`, `déplace` et `affiche`. On constate que, lorsque l'on a instancié un objet de ce type, il est nécessaire de faire appel à la méthode `initialise` pour donner des valeurs à ses attributs. Si, par mégarde, on procède ainsi :

```
Point p
p := Création Point
p.déplace (3, 5)
```

les valeurs des attributs de l'objet référencé par `p` ne seront pas définis au moment de l'appel de `déplace` (certains langages peuvent réaliser des initialisations par défaut mais, même dans ce cas, il n'est pas sûr qu'elles nous conviennent).

Autrement dit, jusqu'ici, il nous fallait compter sur l'utilisateur de l'objet (sous-entendu tout programme utilisant cet objet) pour effectuer l'appel voulu de la méthode `initialise`. La notion de constructeur va permettre de mettre en place un mécanisme d'initialisation automatique, mécanisme qui pourra éventuellement aller au-delà d'une simple attribution de valeurs initiales aux attributs.

D'une manière générale, dans tous les langages objet :

- Un constructeur se présente comme une méthode particulière de la classe, portant un nom conventionnel ; ici, nous conviendrons (comme le font beaucoup de langages) qu'il s'agit du nom de la classe elle-même.
- Un constructeur peut disposer de paramètres.
- Ce constructeur sera appelé au moment de la création de l'objet et il sera possible, le cas échéant, de lui fournir les paramètres souhaités.

5.2 Exemple d'adaptation de notre classe Point

À titre d'exemple, examinons comment faire pour que le travail de la méthode `initialise` de notre classe `Point` du paragraphe 2.3, page 184, soit maintenant réalisé par un constructeur à deux paramètres. Il nous suffira de définir notre nouvelle classe de cette manière :

```
classe Point
{ méthode Point (entier x, entier y)    // constructeur (même nom que la classe)
  { abs := x
    ord := y
  }
  méthode déplace (entier dx, entier dy)
  { abs := abs + dx
    ord := ord + dy
  }
  méthode affiche
  { écrire «Je suis un point de coordonnées », abs, « », ord
  }
}
```

Une nouvelle classe Point*, dotée d'un constructeur*

Lors de la création d'un objet, nous devrons prévoir les paramètres pour ce constructeur. Nous conviendrons de procéder ainsi :

```
p := Création Point (3, 5)    // Allocation de l'emplacement pour un point
                              // et appel du constructeur auquel on fournit
                              // les paramètres 3 et 5
```

Voici comment nous pourrions adapter notre exemple du paragraphe 2.3, page 184 pour qu'il utilise cette nouvelle classe :

```
Point p, r
p := Création Point (3, 5)
p.affiche()
p.deplace(2, 0)
p.affiche()
r := Création Point (6, 8)
r.affiche()
```

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

Exemple d'utilisation de notre nouvelle classe Point

Remarques

1 Il est très important de noter que l'instanciation d'un objet, réalisée par un appel tel que :

```
Création Point (3,5)
```

réalise deux opérations :

- allocation d'un emplacement mémoire pour un objet de type `Point` ;
- appel éventuel du constructeur pour cet objet.

Ces deux opérations sont indissociables. Le constructeur ne peut pas être appelé directement (sur un objet existant), en court-circuitant la première opération :

```
Point p
p := Création Point (...)
  .....
p.Point (...)   // interdit
```

2 Dans nos exemples, le constructeur servait à donner des valeurs initiales aux attributs d'un objet. Il en ira souvent ainsi mais, il faut bien comprendre qu'un constructeur peut très bien réaliser d'autres actions, par exemple : allocation d'emplacements dynamiques, vérification d'existence de fichier, ouverture d'une connexion Internet...

5.3 Surdéfinition du constructeur

Nous avons vu paragraphe 4.6 du chapitre 8, page 162, qu'il est possible de surdéfinir des fonctions. Cette possibilité s'applique également aux méthodes d'une classe, ainsi qu'à son constructeur. Nous pourrons donc définir plusieurs constructeurs se distinguant par le nom-

bre et le type de leurs paramètres. Voici un exemple de définition d'une classe comportant trois constructeurs, accompagné d'un petit exemple d'utilisation :

```
point p, q, r
p := Création Point                // appel constructeur 1
p.affiche
q := Création Point(5)             // appel constructeur 2
q.affiche
r := Création Point(3, 12)         // appel constructeur 3
c.affiche

classe Point
{  méthode Point                          // constructeur 1 (pas de paramètre)
   {  abs := 0
     ord := 0
   }
   méthode Point (entier x)               // constructeur 2 (un paramètre)
   { abs := x
     ord := 0
   }
   méthode Point (entier x, entier y)     // constructeur 3 (deux paramètres)
   { abs := x
     ord := y
   }
   méthode affiche
   { écrire «Je suis un point de coordonnées », abs, « », ord
   }
   entier abs
   entier ord
```

```
Je suis un point de coordonnées 0 0
Je suis un point de coordonnées 5 0
Je suis un point de coordonnées 3 12
```

Exemple de surdéfinition d'un constructeur

5.4 Appel automatique du constructeur

À partir du moment où l'on dote une classe d'un ou plusieurs constructeurs, on peut raisonnablement penser que l'on souhaite qu'un objet ne puisse plus être instancié sans que l'un de ces constructeurs ne soit appelé. C'est bien ce qui est prévu dans la plupart des langages objet. Considérons alors cette classe :

```
classe Point
{ Point (entier x, entier y)  // unique constructeur à 2 paramètres
  { ..... }
```

et cette déclaration :

```
Point p
```

Alors l'instruction suivante sera interdite :

```
p := Creation Point   // interdit
```

En revanche, si la classe `Point` disposait d'un constructeur sans paramètres, l'instruction précédente serait correcte et elle appellerait bien ce constructeur.

Autrement dit, cette instruction d'instanciation est acceptable :

- soit lorsque la classe ne dispose d'aucun constructeur (c'est ce qui se produisait dans les premiers exemples de ce chapitre),
- soit lorsque la classe dispose d'un constructeur sans paramètres (elle peut, bien sûr, en posséder d'autres...).

> Lorsqu'une classe dispose d'au moins un constucteur, il n'est plus possible d'instancier un objet, sans qu'il y ait appel de l'un des constructeurs.

Remarques

1 À l'instar de ce qui se passe dans la plupart des langages, nous n'avons pas prévu de mécanisme permettant à un constructeur de fournir un résultat.

2 Nous avons vu qu'il était possible d'initialiser des variables locales lors de leur déclaration. Il en va de même pour les variables locales des méthodes. En revanche, nous ne prévoierons aucun mécanisme permettant d'initialiser les attributs d'un objet ; par exemple, nous conviendrons que ceci est interdit :

```
class X
{ entier n := 5 ;    // interdit
  .....
}
```

Certains langages autorisent cette possibilité. Il faut alors bien réaliser qu'elle interfère avec le travail du constructeur (qui risque, lui aussi, d'attribuer une valeur à l'attribut `n`). Il est donc nécessaire de savoir dans quel ordre sont effectuées, ces initialisations d'une part, l'appel du constructeur d'autre part.

5.5 Exemple : une classe Carré

Nous vous proposons un exemple exploitant à la fois :

- la surdéfinition du constructeur ;
- la possibilité de modifier l'implémentation d'une classe en conservant son contrat.

Il s'agit de deux implémentations différentes d'une classe `Carré`, dont l'interface serait la suivante :

```
Carré (entier)              // constructeur à un paramètre : le côté du carré
Carré                       // constructeur sans paramètre ; côté 10 par défaut
entier méthode taille       // fournit la valeur du côté
entier méthode surface      // fournit la surface du carré
entier méthode périmètre    // fournit le périmètre du carré
méthode changeCôté (entier) // modifie la valeur du côté du carré
```

Voici une première implémentation qui prévoit tout naturellement un attribut nommé côté, destiné à contenir la valeur du côté du carré :

```
classe Carré
{ méthode Carré (entier n)
  { côté := n
  }
  méthode Carré
  { côté := 10
  }
  entier méthode taille
  { retourne côté
  }
  méthode changeCôté (entier n)
  { côté := n
  }
  entier méthode surface
  { entier s
    s := côté * côté
    retourne s
  }
  entier méthode périmètre
  { entier p
    p : = 4 * côté
    retourne p
  }
  entier côté
}
```

Une implémentation naturelle de la classe Carré

Mais, voici maintenant une seconde implémentation qui prévoit d'autres attributs, à savoir le périmètre et la surface. Cette fois, on voit que les méthodes périmètre et surface deviennent de simple méthodes d'accès et qu'elles n'ont plus à effectuer de calcul à chaque appel. En revanche, ces calculs sont effectués par les méthodes qui sont susceptibles de modifier la valeur du côté, soit ici les constructeurs et changeCôté ; pour simplifier un peu l'écriture, nous avons prévu deux méthodes privées (calculPérimètre et calculSurface) dont l'usage est réservé aux méthodes de la classe :

```
classe Carré
{ méthode carré (entier n)
  { côté := n
    calculPérimètre
    calculSurface
  }
  méthode Carré
  { côté := 10
    calculPérimètre
    calculSurface
  }
```

```
    entier méthode taille
    { retourne côté
    }
    méthode changecôté (entier n)
    { côté := n
      calculPérimètre
      calculSurface
    }
    entier méthode surface
    { retourne surface
    }
    entier méthode périmètre
    { retourne périmètre
    }
    privé méthode calculSurface
    { surface := côté * côté
    }
    privé méthode calculpérimètre
    { périmètre := 4 * côté
    }
    entier côté
    entier surface
    entier périmètre
}
```

Une autre implémentation de la même classe Carré

Cet exemple, certes quelque peu artificiel, montre que, tant qu'on en respecte l'interface, on peut modifier à volonté l'implémentation d'une classe.

Exercice 9.3 Écrire une classe nommée `Carac`, permettant de conserver un caractère, Elle disposera :

- d'un constructeur à un paramètre fournissant le caractère voulu ;

- d'un constructeur sans paramètre qui attribuera par défaut la valeur « espace » au caractère ;

- d'une méthode nommée `estVoyelle` fournissant la valeur `vrai` lorsque le caractère concerné est une voyelle et la valeur `faux` dans le cas contraire.

Écrire un petit programme utilisant cette classe.

Exercice 9.4 Écrire une classe `Rectangle` disposant :

- de trois constructeurs : le premier sans paramètre créera un rectangle dont les deux dimensions sont égales à 1 ; le second à un paramètre sera utilisé à la fois pour les deux dimensions, considérées comme égales ; le troisième à deux paramètres correspondant aux deux dimensions du rectangle. Les dimensions seront de type `réel` ;

- d'une méthode `périmètre` fournissant en résultat le périmètre du rectangle ;

- d'une méthode `surface` fournissant en résultat la surface du rectangle ;

- d'une méthode `agrandit` disposant d'un paramètre de type `réel` correspondant à la valeur par laquelle il faut multiplier les dimensions du rectangle.

Écrire un petit programme d'utilisation.

Exercice 9.5 Écrire une classe nommée `Réservoir`, implémentant l'interface et le « contrat » suivants :

```
méthode Réservoir (entier n)     // crée un réservoir de capacité maximale n
entier méthode verse (entier q)  // ajoute la quantité q au réservoir si possible
                                 // sinon, on ne verse que ce qui est possible
                            // fournit en résultat la quantité réellement ajoutée
entier méthode puise (entier q)  // puise la quantité q si possible
                                 // sinon, on puise le reste
                            // fournit en résultat la quantité réellement puisée
entier méthode jauge             // fournit le «niveau» du réservoir
```

6 Mode des gestion des objets

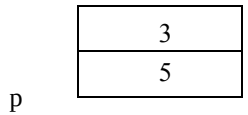
Au paragraphe 2.1, nous vous avons présenté la démarche la plus répandue et la plus souple de «gestion des objets». Nous avons vu qu'elle revient à dissocier ce que nous avons appelé la « variable de type objet » (destinée à contenir une référence à un objet) de l'objet lui-même. Cette variable s'apparente aux variables que nous avions rencontrées jusqu'ici et son emplacement est géré de la même manière (mémoire statique pour les variables du programme principal, pile pour les variables locales aux fonctions ou méthodes). En revanche, l'objet voit son emplacement alloué dynamiquement au moment de l'exécution de l'appel de `Création`. Nous parlerons dorénavant de gestion par référence (on rencontre parfois « sémantique référence ») pour qualifier ce mode de gestion des objets que nous continuerons à privilégier dans les prochains chapitres.

Certains langages objet offrent un autre mode de gestion des objets, que nous nommerons gestion par valeur (on rencontre aussi « sémantique valeur ») qui, en général, cohabite avec le mode précédent. Il consiste à considérer que la seule déclaration d'un objet entraîne la réservation de l'emplacement mémoire correspondant[1]. Si la classe comporte un constructeur, la déclaration de l'objet doit alors préciser des paramètres pour ce constructeur. Ces déclarations se présentent alors sous une forme voisine de :

```
Point p (3, 5)        // réserve l'emplacement pour un objet de type Point
       // et appelle un constructeur, en lui fournissant les paramètres 3 et 5
```

1. C'est d'ailleurs cette hypothèse que nous avions faite pour les tableaux et nous avions écarté alors le cas des tableaux dynamiques existant dans certains langages.

Dans ces conditions, la notation `p` désigne directement l'objet lui-même, ce qu'on peut schématiser ainsi :



À ce niveau, la différence entre les deux modes de gestion des objets peut vous paraître assez minime, la première semblant introduire simplement une variable intermédiaire supplémentaire. En fait, jusqu'ici, nous nous sommes contentés d'utilisations simples. Mais, dans les chapitres suivants, tout en continuant à privilégier la gestion par référence, nous aurons l'occasion de comparer ces deux modes de gestion dans des situations plus complexes.

Pour l'instant, retenez simplement que, dans le premier mode de gestion, on manipule des références à des objets, alors que dans le second, on manipule directement ces objets, c'est-à-dire leur valeur, autrement dit les valeurs de leurs attributs.

Côté langages

Définition d'une classe

Comme vous le verrez dans les exemples de programmes ci-après, la syntaxe de définition d'une classe en C++, C#, Java ou PHP[a] est très proche de celle que nous avons introduite. Dans les quatre langages :

- Les attributs et les méthodes peuvent être soit privés (`private`), soit publics (`public`). Il est donc possible de « violer » le principe d'encapsulation. Certains langages proposent un statut par défaut (`private` pour C++, `public` pour PHP).
- On dispose de constructeurs. En C++, C# ou Java, ils portent le même nom que la classe et ils peuvent être surdéfinis. En PHP, le constructeur se nomme `__construct` ; il ne peut pas être surdéfini, mais il est possible de prévoir des « valeurs par défaut » des paramètres.

Quelques petites différences apparaîtront :

- en C++, on distinguera souvent la déclaration de la classe de la définition de ses méthodes (voyez l'exemple C++ ci-après) ;
- en C++, on utilisera « l'attribut » `private` ou `public`, non pas pour une seule déclaration, mais pour un ensemble de déclarations ;

a.C ne dispose pas de la notion de classe.

- en PHP, les attributs, comme les noms de variable, doivent commencer par $; en outre, un attribut nommé par exemple `$ab` devra être désigné dans une méthode par `$this->abs` et non par `$abs` (qui représenterait alors une variable locale à ladite méthode[a]).

Utilisation d'une classe

Java, C# et PHP utilisent une gestion par référence dans laquelle on utilise `new` (au lieu de `Création`) pour instancier des objets :

```
Point p ;  // ($p en PHP)   : p est une référence sur un objet de type Point
   ...
p = new Point (3, 5) ; // crée un objet de type Point, en appelant un constructeur
```

C++ utilise une gestion par valeur, dans laquelle la déclaration d'un objet en provoque la création :

```
Point p (3, 5) ;  // crée un objet p de type Point, en appelant un constructeur
```

Mais on peut également, en C++ créer dynamiquement des objets, en utilisant des pointeurs qui jouent alors le rôle de nos références :

```
Point *adp ;  // adp est un pointeur contenant l'adresse d'un objet de type Point
   ...
adp = new Point (3,5) ; // crée un objet de type Point, en appelant un constructeur
```

a.Cette complication est due, en partie, au fait que PHP ne déclare pas les types des variables.

Exemples langage

Voici comment se présenterait, dans chacun des langages C++, Java, C# et PHP, notre exemple du paragraphe 5.3, page 190, à savoir une classe `Point`, dotée des trois constructeurs, d'une méthode `deplace` et d'une méthode `affiche`.

Java

En Java, en principe, un fichier source ne contient qu'une seule classe dont il doit porter le nom. Il est cependant possible d'y placer plusieurs classes mais, dans ce cas, une seule d'entre elles (déclarée avec l'accès `public`) sera utilisable. Les autres classes ne seront accessibles qu'à la classe principale (déclarée `public`) du fichier. C'est ainsi que nous avons procédé : la classe principale `TstPoint` contient la méthode `main` et utilise la classe `Point`, « cachée » dans le fichier. Dans un programme véritable, on créerait deux fichiers distincts, l'un nommé `TstPoint`, contenant la classe `TstPoint` (déclarée publique), l'autre nommé `Point` contenant la classe `Point`, déclarée alors publique.

```
public class TstPoint  // classe contenant la méthode principale
{ public static void main (String args[])     // méthode principale
  { Point p = new Point () ;
   p.affiche() ;
   Point q = new Point (3) ;
   q.affiche() ;
   q.deplace (3, 6) ;
   q.affiche() ;
   Point r = new Point (5, 8) ;
   r.affiche() ;
  }
}
     // définition de la classe Point
class Point
{ public Point()                // premier constructeur (sans paramètre)
  { abs = 0 ; ord = 0 ; }
  public Point (int x)          // deuxième constructeur (un paramètre)
  { abs = x ; ord = 0 ; }
  public Point (int x, int y)   // troisième constructeur (deux paramètres)
  { abs = x ; ord = y ; }
  public void deplace (int dx, int dy)    // la méthode deplace
  { abs += dx ; ord += dy ; }
  public void affiche ()                  // la méthode affiche
  {  System.out.println ("Je suis un point de coordonnées " + abs + " " + ord)  ;
  }
  private int abs, ord ;         // il faut préciser private pour encapsuler
}
```

```
Je suis un point de coordonnées 0 0
Je suis un point de coordonnées 3 0
Je suis un point de coordonnées 6 6
Je suis un point de coordonnées 5 8
```

C#

Un fichier source peut contenir une ou plusieurs classes. Ici, nous avons placé la classe `Point` et la classe `tstPoint` contenant la méthode principale (`Main`), dans un même fichier.

```
using System ;
class tstPoint     // classe contenant la méthode principale
{ static void Main()
  { Point p = new Point () ;
    p.affiche() ;
    Point q = new Point (3) ;
    q.affiche() ;
```

```
    q.deplace (3, 6) ;
    q.affiche() ;
    Point r = new Point (5, 8) ;
    r.affiche() ;
  }
}
class Point
{ public Point()               // premier constructeur (sans paramètre)
  { abs = 0 ; ord = 0 ; }
  public Point (int x)         // deuxième constructeur (un paramètre)
  { abs = x ; ord = 0 ; }
  public Point (int x, int y)  // troisième constructeur (deux paramètres)
  { abs = x ; ord = y ; }
  public void deplace (int dx, int dy)    // la méthode deplace
  { abs += dx ; ord += dy ; }
  public void affiche ()                  // la méthode affiche
  {  System.Console.WriteLine ("Je suis un point de coordonnées "
          + abs + " " + ord)  ;
  }
  private int abs, ord ;       // private pour encapsuler
}
```

PHP

On trouve en PHP, un programme principal classique (qui, comme en simple programmation procédurale, n'est ni une fonction, ni une méthode). La définition d'une classe peut figurer dans le même fichier source (comme nous l'avons fait ici), ou dans un fichier séparé qui doit alors être incorporé par une instruction `require`.

Comme nous l'avons indiqué, PHP ne permet de définir qu'un seul constructeur. Il est cependant possible de prévoir, dans son en-tête, des valeurs par défaut pour ses paramètres (ici, 0), lesquelles sont utilisées en cas d'absence dans l'appel. Par ailleurs, dans une méthode, un nom d'attribut se note d'une manière un peu particulière (par exemple $this->abs pour l'attribut `$abs`). Le mot `public` n'est pas indispensable devant les en-têtes de méthodes (elles seront publiques par défaut). En revanche, la déclaration des attributs doit spécifier `public` ou `private` (sinon, comme la déclaration ne comporte pas de nom de type, l'interpréteur ne la « comprend » pas).

```
<?php
$p = new Point () ;    // utilisera la valeur par défaut pour les 2 paramètres
$p->affiche() ;
$q = new Point (3) ;   // utilisera 3 pour le premier paramètre,
                       // la valeur par défaut (0) pour le second
```

```
$q->affiche() ;
$q->deplace (3, 6) ;
$q->affiche() ;
$r = new Point (5, 8) ;
$r->affiche() ;

class point
{ public function __construct ($x = 0, $y = 0) // valeur 0 par défaut pour $x et $y
  { $this->abs = $x ; $this->ord = $y ;        // notez $this->...
  }
  public function deplace ($dx, $dy)
  { $this->abs += $dx ; $this->ord += $dy ; }
  public function affiche ()
  {  echo "Je suis un point de coordonnées ", $this->abs, " ", $this->ord, "<br>"  ;
  }
  private $abs, $ord ;    // mode d'accès obligatoire ici
}
?>
```

```
Je suis un point de coordonnées 0 0
Je suis un point de coordonnées 3 0
Je suis un point de coordonnées 6 6
Je suis un point de coordonnées 5 8
```

C++

Rappelons qu'en C++, par défaut, la gestion des objets est réalisée par valeur. Une simple déclaration telle que :

```
Point p(3,5)
```

réserve l'emplacement pour un objet de type `Point` et appelle le constructeur.

Généralement, pour une classe donnée, on distingue ce que l'on nomme :

- la déclaration d'une classe, laquelle comporte les en-têtes des méthodes (nommées souvent fonctions membres en C++) et la déclaration des attributs (nommés souvent membres données) ;
- la définition des méthodes.

La compilation de la définition des méthodes d'une classe donnée ou la compilation d'un programme utilisant une classe donnée nécessite d'en connnaître la déclaration. Généralement, celle-ci est placée, une fois pour toutes, dans un fichier d'extension `.hpp`, qui est incorporé pour la compilation par une « directive » `#include` appropriée. La définition de la classe, quant à elle, est compilée une fois pour toutes et fournie sous forme d'un module objet qui sera utilisé par l'éditeur de liens pour constituer le programme exécutable.

Toutefois, un même fichier source peut contenir autant de classes qu'on le désire, ainsi qu'éventuellement la fonction principale (`main`). Ici, dans notre exemple, par souci de simplicité, nous avons placé dans un même fichier source la déclaration de la classe, sa définition et le programme l'utilisant.

```
#include <iostream>
using namespace std ;
    // déclaration de la classe Point
class Point
{ public :
    Point() ;                   // premier constructeur (sans paramètres)
    Point (int) ;               // deuxième constructeur (un paramètre)
    Point (int, int);           // troisième constructeur (deux paramètres)
    void deplace (int, int) ;
    void affiche () ;
  private :
   int abs, ord ;
} ;           // attention à ce point-virgule
   // définitions des méthodes de la classe Point (leur compilation nécessite
   //     la déclaration de la classe, fournie ici auparavant, dans le fichier)
Point::Point ()
{ abs = 0 ; ord = 0 ; }
Point::Point (int x)
{ abs = x ; ord = 0 ; }
Point::Point (int x, int y)
{ abs = x ; ord = y ; }
void Point::deplace (int dx, int dy)
{ abs += dx ; ord += dy ; }
void Point::affiche ()
{  cout << "Je suis un point de coordonnées "<< abs << " " << ord << "\n"  ;
}
  // programme principal ; sa compilation nécessite la déclaration
  //     de la classe Point (fournie ici auparavant)
 main()
{ Point p  ;
  p.affiche() ;
  Point q (3) ;
  q.affiche() ;
  q.deplace (3, 6) ;
  q.affiche() ;
  Point r (5, 8) ;
  r.affiche() ;
}
```

```
Je suis un point de coordonnées 0 0
   Je suis un point de coordonnées 3 0
   Je suis un point de coordonnées 6 6
   Je suis un point de coordonnées 5 8
```

À titre indicatif, voici une autre version de ce même programme utilisant une gestion dynamique des objets, et non plus une gestion par valeur. La différence porte essentiellement sur la syntaxe, au niveau de la fonction principale (la déclaration et la définition de la classe restant inchangées). La durée de vie des objets est ici quasiment la même dans les deux cas : dans le premier exemple, il s'agissait d'objets locaux à la fonction `main`, alors qu'ici il s'agit d'objets créés dynamiquement dans cette même fonction.

```
main()
{ Point *adp, *adq, *adr  ;  // adp, adq et adr sont
                             //   des pointeurs sur un objet de type Point
  adp = new Point () ;       // création dynamique d'un objet de type Point
  (*adp).affiche() ;         // ou   adp->affiche() ;
  adq = new Point (3) ;
  (*adq).affiche() ;         // ou   adq->affiche() ;
  (*adq).deplace (3, 6) ;    // ou   adq->deplace (3, 6) ;
  (*adq).affiche() ;         // ou   adq->affiche() ;
  adr = new Point (5, 8) ;   // Création dynamique d'un autre objet de type Point
  (*adr).affiche() ;         // ou   adr->ffiche() ;
}
```

Signalons enfin qu'il est possible d'utiliser une syntaxe de définition de classe, voisine de celles des autres langages, en fournissant directement les définitions des méthodes, comme dans ce canevas :

```
class Point
{ public :
    Point()  { abs = 0 ; ord = 0 ; }
    Point (int) { abs = x ; ord = 0 ; }
         .....
    void deplace (int, int) { abs += dx ; ord += dy ; }
         .....
  private :
   int abs, ord ;
} ;
```

Mais il faut savoir que les méthodes ainsi définies dans la déclaration de la classe sont ce que l'on nomme des « méthodes en ligne », ce qui signifie que le compilateur peut incorporer les instructions correspondantes dans le programme, à chaque fois qu'on les appelle (on n'a donc plus affaire à un mécanisme de fonction). Il s'agit d'une technique qui optimise le temps d'exécution, au détriment de la place mémoire.