Claude Delannoy

S'initier à la programmation et à l'orienté objet

Deuxième édition 2014

Avec des exemples en C, C++, C#, Java, Python et PHP

EYROLLES

 ISBN : 978-2-212-14067-5

Avant-propos

Objectif de l'ouvrage

Ce livre se propose de vous apprendre à programmer en exprimant les concepts fondamentaux à l'aide d'un « pseudo-code ». Cela vous permet de rédiger des programmes en privilégiant l'aspect algorithmique, sans être pollué par la complexité et la technicité d'un langage donné. Par ailleurs, l'ouvrage montre comment ces concepts fondamentaux se traduisent dans six langages très usités (C, C++, Java, C#, Python et PHP) et fournit des exemples complets. Il prépare ainsi efficacement à l'étude d'un langage réel.

Forme de l'ouvrage

L'ouvrage a été conçu sous forme d'un cours, avec une démarche très progressive. De nombreux exemples complets, écrits en pseudo-code et accompagnés du résultat fourni par leur exécution, viennent illustrer la plupart des concepts fondamentaux. Des exercices appropriés proposent la rédaction de programmes en pseudo-code, permettant ainsi la mise en pratique des acquis. Plutôt que de les regrouper classiquement en fin de chapitre, nous avons préféré les placer aux endroits jugés opportuns pour leur résolution. Une correction est fournie en fin de volume ; nous vous encourageons vivement à ne la consulter qu'après une recherche personnelle et à réfléchir aux différences de rédaction qui ne manqueront pas d'apparaître.

Chaque chapitre se termine par :

- Une rubrique « Côté langages » qui montre comment les concepts exposés préalablement s'expriment dans les six langages choisis ; elle constitue une sorte de guide de traduction du

pseudo-code dans un véritable langage. Notez que le langage C n'étant pas orienté objet, il n'est pris en compte que jusqu'au chapitre 8.

- Une rubrique « Exemples langages » qui propose des programmes complets, traduction de certains des exemples présentés en pseudo-code.

À qui s'adresse cet ouvrage

Cet ouvrage s'adressse aux débutants en programmation et aux étudiants du premier cycle d'université. Il peut également servir :

- à ceux qui apprennent à programmer directement dans un langage donné : il leur permettra d'accompagner leur étude, en dégageant les concepts fondamentaux et en prenant un peu de recul par rapport à leur langage ;
- à ceux qui maîtrisent déjà la programmation dans un langage donné et qui désirent « passer à un autre langage » ;
- à ceux qui connaissent déjà la programmation procédurale et qui souhaitent aborder la programmation orientée objet.

Enfin, sa conception permet à l'ouvrage d'être facilement utilisé comme « support de cours ».

Plan de l'ouvrage

Le **chapitre 1** présente le rôle de l'ordinateur, les grandes lignes de son fonctionnement et la manière de l'utiliser. Il dégage les importantes notions de langage, de programme, de données et de résultats, de système d'exploitation et d'environnement de programmation.

Le **chapitre 2** introduit les concepts de variable et de type, et la première instruction de base qu'est l'affectation. Il se limite à trois types de base : les entiers, les réels et les caractères. Il présente les erreurs susceptibles d'apparaître dans l'évaluation d'une expression et les différentes façons dont un langage donné peut les gérer. On y inclut les notions d'expression mixte et d'expression constante.

Le **chapitre 3** est consacré aux deux autres instructions de base que sont la lecture et l'écriture. Il nous a paru utile de les placer à ce niveau pour permettre, le plus rapidement possible, de présenter et de faire écrire des programmes complets. On situe ces instructions par rapport aux différents modes de communication entre l'utilisateur et le programme : mode console, programmation par événements, mode batch, programmation Internet.

Le **chapitre 4** étudie la structure de choix, en présentant la notion de condition et en introduisant le type booléen. On y aborde les choix imbriqués. L'existence de structures de choix multiple (instruction `switch` des six langages examinés) est évoquée dans la partie « Côté langages ».

Le **chapitre 5** aborde tout d'abord les structures de répétition conditionnelle. Il présente la notion de compteur, avant d'examiner les structures de répétition inconditionnelle (ou « avec compteur ») et les risques inhérents à la modification intempestive du compteur.

Le **chapitre 6** présente les « algorithmes élémentaires » les plus usuels : comptage, accumulation, recherche de maximum, imbrication de répétitions. Il donne un aperçu de ce qu'est l'itération.

Le **chapitre 7** traite des tableaux, à une ou deux dimensions. Il se place a priori dans un contexte de gestion statique des emplacements mémoire correspondants et il décrit les contraintes qui pèsent alors sur la taille d'un tableau. Les autres modes de gestion (automatique et dynamique) sont néanmoins évoqués en fin de chapitre, ainsi que la notion de tableau associatif (utilisé par exemple par PHP) qui est comparée à celle de tableau indicé. Les situations de débordement d'indice sont examinées, avec leurs conséquences potentielles dépendantes du langage.

Le **chapitre 8** est consacré aux fonctions. Il présente les notions de paramètres, de variable locale et de résultat, et distingue la transmission par valeur de la transmission par référence (par adresse), en examinant le cas particulier des tableaux. Il aborde la durée de vie des variables locales, ce qui amène à traiter du mode de gestion automatique correspondant (et du concept de pile qu'il utilise souvent). Il dégage le concept de « programme principal » ou de « fonction principale ». Enfin, il donne un aperçu de ce qu'est la récursivité.

Le **chapitre 9** introduit les notions de classe, d'attribut, d'objet, de méthode, d'encapsulation des données et de constructeur. Il fournit quelques éléments concernant les deux modes de gestion possibles des objets, à savoir par référence ou par valeur. Il étudie les possibilités d'amendement du principe d'encapsulation par modification des droits d'accès aux attributs ou aux méthodes.

Le **chapitre 10** examine l'incidence du mode de gestion des objets (par référence ou par valeur) sur l'affectation d'objets et sur la durée de vie des objets locaux. Il aborde les objets transmis en paramètre et il convient, comme c'est le cas dans la plupart des langages objet, que « l'unité d'encapsulation est la classe et non l'objet ». Il analyse le cas des objets fournis en résultat. Puis, il étudie les attributs et les méthodes de classe, et traite sommairement des tableaux d'objets et des problèmes qu'ils posent dans l'appel des constructeurs, ainsi que des situations « d'auto-référence ».

Le **chapitre 11** est consacré à la composition des objets, c'est-à-dire au cas où un attribut d'une classe est lui-même de type classe. Il examine les problèmes qui peuvent alors se poser au niveau des droits d'accès et dans la nature de la relation qui se crée entre les objets concernés. Il présente la distinction entre copie profonde et copie superficielle d'un objet. Il montre également comment résoudre un problème fréquent, à savoir réaliser une classe à instance unique (singleton).

Le **chapitre 12** présente la notion d'héritage ou de classe dérivée et son incidence sur les droits d'accès aux attributs et aux méthodes. Il fait le point sur la construction des objets dérivés avant de traiter de la redéfinition des méthodes. Il aborde les situations de dérivations

successives et décrit succinctement les possibilités de modification des droits d'accès lors de la définition d'une classe dérivée.

Le **chapitre 13** expose les notions de base du polymorphisme, à savoir la compatibilité par affectation et la ligature dynamique. Il en examine les conséquences dans plusieurs situations et montre quelles sont les limites de ce polymorphisme, ce qui conduit, au passsage, à parler de valeurs de retour covariantes présentes dans certains langages.

Le **chapitre 14** traite enfin de concepts moins fondamentaux que l'héritage ou le polymorphisme, parfois absents de certains langages, mais qui peuvent faciliter la conception des logiciels. Il s'agit des notions de classes abstraites (ou retardées), d'interface et d'héritage multiple.

Justifications de certains choix

Voici quelques éléments justifiant les choix que nous avons opéré dans la conception de cet ouvrage.

- Nous présentons la programmation procédurale avant d'introduire la programmation objet, pour différentes raisons :
  - la plupart des langages objet actuels offrent des possibilités de programmation procédurale ;
  - la programmation orientée objet s'appuie sur les concepts de la programmation procédurale ; la seule exception concerne la notion de fonction indépendante qui peut être absente de certains langages objet mais qui se retrouve quand même sous une forme très proche dans la notion de méthode ;
  - sur un plan pédagogique, il est difficile d'introduire directement des méthodes dans une classe si l'on n'a pas encore étudié l'algorithmique procédurale.
- Dans le choix des concepts fondamentaux, nous avons évité de nous limiter à un sous-ensemble commun à tous les langages car cela aurait été trop réducteur à notre sens. Nous avons choisi les concepts qu'il nous a paru important de maîtriser pour pouvoir ensuite aborder la programmation dans n'importe quel langage.

  Ces choix font ainsi du pseudo-code, non pas la « matrice » de tous les langages, mais plutôt un langage à part entière, simple, mais renfermant la plupart des concepts fondamentaux de la programmation – certains pouvant ne pas exister dans tel ou tel langage. C'est précisément le rôle de la partie « Côté langages » que de monter en quoi les langages réels peuvent différer les uns des autres et ce au-delà de leur syntaxe (les six langages choisis possèdent la même syntaxe de base et jouissent pourtant de propriétés différentes). Ainsi, le lecteur est non seulement amené à programmer en pseudo-code mais, en même temps, il est préparé à affronter un vrai langage. Voici par exemple quelques points sur lesquels les langages peuvent se différencier les uns des autres :

  - mode de traduction : compilation (C, C++), interprétation (PHP ou Python) ou traduction dans un langage intermédiaire (Java) ;
  - mode de gestion de la mémoire : statique, automatique, dynamique ;
  - nature des expressions constantes et des expression mixtes ;
  - gestion des tableaux (sous forme indicée ou sous forme de tableau associatif comme en PHP) ;
  - mode de transmission des paramètres d'une fonction : par valeur, par référence ;
  - utilisation pour les objets d'une « sémantique de valeur » ou d'une « sémantique de référence » (Java n'utilise que la deuxième, tandis que C++ utilise les deux) ;
  - mode standard de recopie des objets : copie superficielle ou copie profonde.
- Nous n'avons pas introduit de type chaîne car son implémentation varie fortement suivant les langages (type de base dans certains langages procéduraux, type hybride en C, type classe dans les langages objet...). Sa gestion peut se faire par référence ou par valeur. Dans certains langages, ces chaînes sont constantes (non modifiables), alors qu'elles sont modifiables dans d'autres...

Table des matières

5

Les structures de répétition

Après avoir vu comment réaliser des structures de choix, nous allons maintenant étudier les structures les plus puissantes de la programmation, à savoir les structures de répétition (ou de boucle) : elles permettent d'exécuter à plusieurs reprises une suite d'instructions. Dans la plupart des langages, ces répétitions se classent en deux catégories :

- Les répétitions conditionnelles (ou « indéfinies ») : la poursuite de la répétition des instructions concernées dépend d'une certaine condition qui peut être examinée :
  - soit après les instructions à répéter : on parle généralement de répétition jusqu'à ;
  - soit avant les instructions à répéter : on parle généralement de répétition tant que.
- Les répétitions inconditionnelles (ou « avec compteur » ou « définies ») : les instructions concernées sont répétées un nombre donné de fois.

Nous commnencerons par examiner les deux sortes de répétitions conditionnelles. Nous introduirons ensuite la notion de compteur et nous verrons comment la mettre en œuvre pour programmer une répétition inconditionnelle. Nous verrons enfin comment la répétition avec compteur permet d'exprimer plus simplement les choses.

1 La répétition jusqu'à

1.1 Exemple introductif

Considérez ce programme :

```
entier n
répéter
  { écrire «donnez un nombre entier :»
    lire n
    écrire «voici son carré : », n*n
  }
jusqu'à n = 0
écrire «fin du programme»
```

Affichage des carrés de valeurs fournies en données (répétition jusqu'à*)*

Les mots `répéter` et `jusqu'à` encadrent le bloc d'instructions :

```
écrire «donnez un nombre entier :»
lire n
écrire «voici son carré : », n*n
```

Ils signifient que ces instructions doivent être répétées autant de fois qu'il est nécessaire et ceci jusqu'à ce que la condition `n=0` soit vraie.

Notez bien que le nombre de répétitions n'est pas indiqué explicitement. Il dépendra ici des données que l'on fournira au programme. D'autre part, pour chacune de ces répétitions, la condition `n=0` n'est examinée qu'après l'exécution des trois instructions concernées. Pour bien expliciter cela, voyons ce que produit ce programme lorsqu'on lui fournit successivement les valeurs 3, 12, -6 et 0 ; les résultats se présentent ainsi :

```
donnez un nombre entier :
3
voici son carré : 9
donnez un nombre entier :
12
voici son carré : 144
donnez un nombre entier :
0
voici son carré : 0
fin du programme
```

Pour chaque valeur lue en donnée, nous affichons le carré. Vous constatez que nous obtenons effectivement les carrés des nombres 3, 9 et -6, mais il en va de même pour la valeur 0.

En effet, chronologiquement, les trois instructions de la boucle ont été exécutées une première fois, donnant à n la valeur 3 et en affichant le carré. La condition `n=0` a alors été examinée. Étant fausse, les trois instructions ont été répétées à nouveau, donnant alors à `n` la valeur 12 et en affichant le carré. Là encore, la condition `n=0` étant fausse, les instructions ont été répétées une troisième fois, donnant à `n` la valeur 0 et en affichant le carré. Cette fois, la condition `n=0` étant vraie, la répétition a été interrompue et l'on a exécuté l'instruction suivant le mot `jusqu'à`, laquelle a écrit `fin du programme`.

1.2 Nos conventions d'écriture

A priori, les instructions régies par la répétition sont celles qui se trouvent entre les mots `répéter` et `jusqu'à`. Nous aurions donc pu nous passer de les placer dans un bloc. Néanmoins, dans d'autres structures de répétition, nous ne trouverons pas de « délimiteur » de fin. Dans ces conditions, par souci d'homogénéité nous préférons utiliser un bloc dès qu'il y a plus d'une instruction.

D'une manière générale, nous conviendrons donc que la répétition `jusqu'à` se note ainsi :

```
répéter
    instruction
jusqu'à condition
```

La répétition jusqu'à

- *instruction* : instruction de base, bloc d'instructions ou instruction structurée telle que choix ou boucle (nous en verrons des exemples par la suite),
- *condition* : expression booléenne.

Notez que, par souci de lisibilité, lorsque nous aurons affaire à un bloc, nous en décalerons les instructions, en les alignant, comme nous l'avons fait dans l'exemple d'introduction.

Remarques

1 Les instructions d'une répétition `jusqu'à` sont toujours répétées au moins une fois (puisque la condition n'est examinée qu'après exécution de ces instructions). On dit souvent qu'on fait toujours au moins un « tour de boucle ». Nous verrons qu'il n'en ira pas de même avec la répétition `tant que`.

2 Si la condition mentionnée ne devient jamais fausse, les instructions de la boucle sont répétées indéfiniment. On dit souvent, dans ce cas, que le programme « boucle ». Il est généralement possible d'interrompre un programme qui boucle, moyennant l'utilisation d'une démarche appropriée (dépendant de l'environnement de programmation utilisé).

Exercice 5.1 Écrire un programme qui demande à l'utilisateur de lui fournir un nombre entier positif et inférieur à 100 et ceci jusqu'à ce que la réponse soit satisfaisante. Le dialogue avec l'utilisateur se présentera ainsi :

```
donnez un entier positif inférieur à 100 :
453
donnez un entier positif inférieur à 100 :
25
merci pour le nombre 25
```

1.3 Exemples

1.3.1 Recherche de la première voyelle d'un mot

Voici un programme qui demande à l'utilisateur de lui fournir un mot et qui en affiche la première voyelle. Nous supposerons que ce mot est écrit en minuscules.

```
caractère c
écrire «donnez un mot écrit en minuscules :»
répéter
   lire c
jusqu'à c='a' ou c='e' ou c='i' ou c='o' ou c='u' ou c='y'
écrire «première voyelle : », c
```

```
donnez un mot écrit en minuscules :
programmation
première voyelle : o
```

Recherche de la première voyelle d'un mot

Notez que nous faisons l'hypothèse que l'utilisateur peut fournir en une seule fois tous les caractères voulus (en les validant), sans espace entre eux. Ils sont ensuite lus individuellement (ici par l'unique instruction `lire c`)

Ici, nous n'avons pas prévu le cas où le mot ne contient aucune voyelle, ce qui peut se produire par exemple, par erreur de frappe. Dans ce cas, le programme va chercher indéfiniment à lire des caractères (et l'utilisateur ne sera pas prévenu de cette demande excédentaire !). Dans la pratique, il faudra être très prudent vis-à-vis de genre de situation et s'assurer qu'une boucle ne peut pas devenir « infinie ». On pourrait penser à demander à l'utilisateur de fournir un caractère supplémentaire pour marquer la fin de son mot (espace, point...). Le programme se compliquerait déjà un peu ; la fin pourrait se présenter ainsi :

```
répéter
   lire c
jusqu'à c='a' ou c='e' ou c='i' ou c='o' ou c='u' ou c='y' ou c = ' '
si c != ' ' alors écrire «première voyelle : », c
           sinon écrire «pas de voyelles»
```

Mais, en toute rigueur, notre boucle ne se terminerait toujours pas si aucun espace (et aucune voyelle) n'apparaît dans les données. Une démarche, plus radicale, pourrait consister à ne considérer qu'un nombre maximal de caractères (30, 80...) ; il faudrait alors utiliser un « compteur », technique que nous apprendrons ultérieurement.

1.3.2 Doublement de capital

Voici un programme qui demande à l'utilisateur de lui fournir la valeur d'un capital qu'il souhaite placer, ainsi que le taux (annuel) auquel sera effectué le placement. Il affiche l'évolution annuelle de ce capital jusqu'à ce qu'il ait atteint ou dépassé le double du capital initial.

Nous utiliserons une variable de type `réel` nommée `cap` qui contiendra la valeur du capital, au fil des différentes années. Elle devra être initialisée avec la valeur fournie par l'utilisateur, et sa progression d'une année à la suivante sera réalisée par une instruction telle que (`taux` désignant la variable contenant le taux du placement) :

```
cap := cap * ( 1 + taux )
```

Notez que la « nouvelle » valeur de `cap` est obtenue par une expression faisant intervenir l'ancienne valeur ; cette instruction est analogue à une instruction telle que `i:=i+1`.

Par ailleurs, comme il est nécessaire de comparer ce capital à un moment donné avec le capital initial, il nous aura fallu prendre soin de conserver cette dernière valeur ; ici, nous utiliserons la variable nommée `capIni`.

Voici le programme complet, accompagné d'un exemple d'exécution (ici, par souci de clarté, nous affichons les nombres réels avec deux chiffres décimaux) :

```
réel capIni, cap, taux
écrire «donnez le capital à placer et le taux : »
lire cap, taux
capIni := cap
répéter
  { cap := cap * (1 + taux)
    écrire «capital un an plus tard : », cap
  }
jusqu'à cap >= 2 * capIni
écrire «fin du programme»
```

```
donnez le capital à placer et le taux :
10000  0.12
capital un an plus tard :     11200.00
capital un an plus tard :     12544.00
capital un an plus tard :     14049.28
capital un an plus tard :     15735.19
capital un an plus tard :     17623.42
capital un an plus tard :     19738.23
capital un an plus tard :     22106.82
fin du programme
```

Évolution annuelle d'un capital jusqu'à ce qu'il ait doublé (1)

Remarque

Nous pourrions remplacer les instructions :

```
  lire cap, taux
  capIni := cap
```

par :

```
lire capIni, taux
cap := capIni
```

1.4 Faire des choix dans une boucle

Comme nous l'avons déjà mentionné, les différentes structures peuvent « s'imbriquer » les unes dans les autres. Voici un exemple de structure de choix imbriquée dans une boucle `jusqu'à`. Il s'agit d'un programme qui recherche et affiche toutes les voyelles d'un mot fourni en minuscules au clavier et terminé par un caractère espace. Là encore, nous faisons l'hypothèse que tous les caractères peuvent être fournis en une seule fois.

```
caractère c
écrire «donnez un mot suivi d'un espace : »
répéter
  { lire c
    si c='a' ou c='e' ou c='i' ou c='o' ou c='u' ou c='y'
        alors écrire c
  }
jusqu'à c = ' '
```

```
donnez un mot suivi d'un espace :
anticonstitutionnellement
aioiuioeee
```

Affichage des voyelles d'un mot

Exercice 5.2 Dans l'exercice 5.1, l'utilisateur se voit poser la même question, qu'il s'agisse d'une première demande ou d'une nouvelle demande suite à une réponse incorrecte. Améliorez-le de façon que le dialogue se présente ainsi :

```
donnez un entier positif inférieur à 100 :
453
SVP, inférieur à 100 :
0
SVP, positif :
25
merci pour le nombre 25
```

2 La répétition tant que

Comme nous l'avons dit en introduction de ce chapitre, les répétitions indéfinies peuvent examiner leur condition, soit en fin de boucle comme la répétition `jusqu'à` que nous venons d'étudier, soit en début de boucle comme la répétition `tant que` que nous allons exposer maintenant.

2.1 Exemple introductif

Voici comment nous pourrions réécrire notre exemple de doublement de capital du paragraphe 1.3.2, en utilisant une répétition `tant que` (les résultats seraient les mêmes) :

```
réel capIni, cap, taux
écrire «donnez le capital à placer et le taux : »
lire cap, taux
capIni := cap
tant que cap < 2 * capIni répéter
  { cap := cap * (1 + taux)
    écrire «capital un an plus tard : », cap
  }
écrire «fin du programme»
```

Évolution annuelle d'un capital jusqu'à ce qu'il ait doublé (2)

Cette fois, l'instruction :

```
tant que cap < 2 * capIni répéter
```

commence par examiner la valeur de la condition `cap < 2 * capIni` ; si elle est vraie, elle exécute les instructions du bloc qui suit :

```
cap := cap * (1 + taux)
écrire «capital un an plus tard : », cap
```

Puis elle examine à nouveau la condition... et ainsi de suite. Lorsque la condition est fausse, on passe à l'instruction suivant le bloc, soit ici l'écriture de `«fin du programme»`.

2.2 Conventions d'écriture

Nous conviendrons qu'une répétition `tant que` se note ainsi :

```
tant que condition répéter
    instruction
```

La répétition tant que

- *instruction* : instruction de base, bloc d'instructions ou instruction structurée (choix ou boucle) ;
- *condition* : expression booléenne.

Remarques

1 De par la nature même de la répétition `tant que`, la condition régissant la répétition est évaluée avant la première exécution. Cette condition doit donc pouvoir être calculée à ce moment, ce qui signifie qu'elle ne peut pas faire intervenir des variables qui ne se trouvent définies que dans la boucle. Nous verrons plus loin un exemple où cet aspect se révèle important.

Dans le cas de la répétition `jusqu'à`, la condition n'étant évaluée qu'en fin de boucle, les variables y intervenant pouvaient très bien n'être définies que lors du premier tour de boucle.

2 Il est possible que la condition régissant la répétition soit fausse dès le début, auquel cas, les instructions de la boucle ne sont pas exécutées (on dit qu'on ne fait aucun « tour de boucle »). Rappelons que les instructions d'une répétition `jusqu'à` étaient toujours exécutées au moins une fois.

3 Là encore, si la condition qui régit la boucle ne devient jamais fausse, les instructions correspondantes sont répétées indéfiniment.

4 Attention à l'erreur usuelle qui consiste à vouloir répéter plusieurs instructions, en omettant de les inclure dans un bloc. Si nous procédions ainsi dans notre exemple précédent :

```
tant que cap < 2 * capIni répéter
   cap := cap * (1 + taux)
   écrire «capital un an plus tard : », cap
```

seule l'instruction d'affectation serait concernée par la répétition (rappelons que nos alignements d'instructions ne sont là que pour rendre plus claires les structures, mais qu'ils ne sont pas « significatifs »). Ce n'est que lorsque le capital aurait doublé que nous passerions à la suite de la boucle, en affichant la dernière valeur du capital obtenue. Notez qu'ici, les conséquences de cette erreur resteraient assez limitées. Dans d'autres circonstances, elles pourraient conduire à des résultats faux ou à une boucle infinie.

2.3 Lien entre répétition tant que et répétition jusqu'à

En fait, des deux structures de répétition conditionnelles que nous venons de présenter, une seule (n'importe laquelle) est indispensable. En effet, le canevas suivant (dans lequel instruction représente une instruction au sens large, c'est-à-dire éventuellement un bloc) :

```
répéter
  instruction
jusqu'à condition
```

est équivalent au suivant :

```
instruction
tant que (non condition) répéter
  instruction
```

De même, le canevas :

```
tant que condition répéter
  instruction
```

est équivalent au suivant :

```
répéter
   si condition alors instruction
jusqu'à condition
```

Remarques

1 D'une manière générale, on démontre que tout programme peut s'écrire en ne faisant appel qu'à deux structures fondamentales : la structure de choix et une seule structure de répétition conditionnelle. La plupart des langages offrent d'avantage de structures, ceci afin de fournir des formulations mieux adaptées à un type de problème. L'exemple suivant vous en fournit une illustration.

2 Nous avons utilisé ici la terminologie la plus courante (`tant que` et `jusqu'à`) en ce qui concerne les répétitions conditionnelles. On peut en rencontrer d'autres : répétition avec test d'arrêt en début, répétition avec test d'arrêt en fin.

2.4 Exemple

Cherchons à réécrire le programme du paragraphe 1.1 à l'aide d'une répétition `tant que`. Si nous utilisons le canevas présenté comme équivalent à une répétition `jusqu'à`, nous aboutirions à cette solution :

```
entier n
écrire «donnez un nombre entier : »
lire n
écrire «voici son carré : », n*n
tant que n != 0 répéter
  { écrire «donnez un nombre entier :»
    lire n
    écrire «voici son carré : », n*n
  }
écrire «fin du programme»
```

Affichage des carrés de valeurs fournies en données (répétition tant que*) (1)*

Cette rédaction paraît peu satisfaisante, dans la mesure où elle renferme deux fois les instructions de lecture d'un nombre et d'affichage de son carré. On peut éviter cela en donnant artificiellement une valeur non nulle à `n`, avant d'aborder la répétition `tant que` :

```
entier n
n := 1
tant que n != 0 répéter
  { écrire «donnez un nombre entier :»
    lire n
    écrire «voici son carré : », n*n
  }
écrire «fin du programme»
```

Affichage des carrés de valeurs fournies en données (répétition tant que*) (2)*

Cet artifice est nécessaire pour que la condition `n != 0` soit définie (et fausse) au moment où l'on aborde la boucle. Bien entendu, d'autres solutions seraient envisageables, par exemple :

```
entier n
booléen fini := faux
tant que non fini répéter
  { écrire «donnez un nombre entier :»
    lire n
    écrire «voici son carré : », n*n
    si n = 0 alors fini := vrai
  }
écrire «fin du programme»
```

Affichage des carrés de valeurs fournies en données (répétition tant que*) (3)*

On notera qu'aucune des deux formulations n'est aussi naturelle que celle utilisant une répétition `jusqu'à`. Ce qui montre bien l'intérêt de pouvoir disposer des deux sortes de boucles.

Exercice 5.3 Réécrire le programme demandé dans l'exercice 5.1 en utilisant une répétition `tant que`.

Exercice 5.4 Réécrire le programme demandé dans l'exercice 5.2 en utilisant une répétition `tant que`.

3 Comment réaliser des répétitions inconditionnelles

Nous venons d'étudier les répétitions conditionnelles, à savoir les répétitions `tant que` et `jusqu'à`. Mais, comme nous l'avons dit en introduction de ce chapitre, une répétition peut être également inconditionnelle, c'est-à-dire que son nombre de tours est parfaitement déterminé. Tous les langages possèdent une telle structure que nous présenterons dans le paragraphe suivant. Mais auparavant, nous allons voir comment la mettre en œuvre à l'aide des instructions que nous avons rencontrées jusqu'ici, ce qui nous amènera à présenter la notion de compteur.

3.1 La notion de compteur de boucle

Il est possible de compter les « tours de boucle » à l'aide d'une variable entière qu'on initialise à 0 et dont on augmente la valeur de 1 à chaque tour. On peut ensuite utiliser ce compteur de deux façons différentes :

- soit simplement pour en exploiter la valeur, aussi bien au sein des instructions de la boucle qu'après la fin de la boucle ; nous parlerons « d'exploitation passive » du compteur ;
- soit pour limiter effectivement le nombre de tours de boucle en introduisant une condition de poursuite faisant intervenir le compteur : nous parlerons « d'exploitation active » du compteur.

Nous allons d'abord examiner la première situation, essentiellement dans le but de vous présenter la technique du comptage que nous appliquerons ensuite à la seconde situation. Nous verrons alors comment cette dernière peut être simplifiée par une instruction appropriée de répétition inconditionnelle.

3.2 Introduire un compteur dans une répétition

3.2.1 Exemple 1

Considérons à nouveau le programme de calcul de carrés du paragraphe 1.1, en supposant que nous souhaitions indiquer à l'utilisateur combien de valeurs ont été traitées. Il nous suffit :

- de déclarer une variable entière servant de compteur que nous nommerons ici `i` ;
- de s'arranger pour que `i` possède la valeur 0 avant d'aborder la boucle ;
- d'augmenter (on dit aussi « d'incrémenter ») la valeur de `i` d'une unité, à chaque parcours de la boucle, en plaçant, parmi les instructions de cette dernière (la place exacte n'ayant ici aucune importance), l'instruction :

```
i := i + 1
```

Voici ce que pourrait être le programme voulu, accompagné d'un exemple d'exécution :

```
entier n           // pour le nombre fourni par l'utilisateur
entier i           // compteur du nombre de valeurs traitées
i := 0
répéter
  { écrire «donnez un nombre entier : »
    lire n
    écrire «voici son carré : », n*n
    i := i + 1
  }
jusqu'à n = 0
écrire «vous avez fourni », i, «valeurs (y compris le 0 de fin)»
```

```
donnez un nombre entier : 5
voici son carré : 25
donnez un nombre entier : 12
voici son carré : 144
donnez un nombre entier : 0
voici son carré : 0
vous avez fourni 3 valeurs (y compris le 0 de fin)
```

Affichage des carrés de valeurs fournies en données avec comptage

Remarque

Nous avons introduit ici quelques indications dans notre programme, précédées des caractères //. Il s'agit de ce que l'on nomme des commentaires, c'est-à-dire du texte destiné au lecteur du programme et n'ayant aucune influence sur sa traduction. Nous conviendrons par la suite que le texte qui suit ces caractères // jusqu'à la fin de la ligne constitue un tel commentaire.

3.2.2 Exemple 2

Dans le précédent programme, nous n'utilisions la valeur du compteur qu'après la fin de la boucle ; il est naturellement possible de l'exploiter également à l'intérieur de la boucle, comme dans cet exemple, dans lequel nous « numérotons » les valeurs demandées à l'utilisateur :

```
entier n           // pour le nombre fourni par l'utilisateur
entier i           // compteur du nombre de valeurs traitées
i := 0
répéter
  { i := i + 1    // attention à l'emplacement de cette incrémentation
    écrire «donnez un », i, «ème nombre entier : »
    lire n
    écrire «voici son carré : », n*n
  }
jusqu'à n = 0
}
écrire «vous avez fourni », i, «valeurs (y compris le 0 de fin)»
```

```
donnez un 1ème nombre entier : 5
voici son carré : 25
donnez un 2ème nombre entier : 12
voici son carré : 144
donnez un 3ème nombre entier : 0
voici son carré : 0
vous avez fourni 3 valeurs (y compris le 0 de fin)
```

Affichage des carrés de valeurs fournies en données avec comptage et numérotation

Remarques

1 Ici, l'emplacement de l'instruction `i := i + 1` est important puisque l'on utilise le compteur `i` dans la boucle. Notez cependant que d'autres constructions sont possibles, par exemple :

```
i := 1                          // initialisation du compteur à 1
répéter
   { écrire «donnez un », i, «ème nombre entier : »
     lire n
     écrire «voici son carré : », n*n
     i := i + 1                 // +1 sur le compteur - emplacement important
   }
jusqu'à n = 0
écrire «vous avez fourni », i-1, «valeurs (y compris le 0 de fin)»
                        // attention : i-1 cette fois
```

Il vous arrivera d'ailleurs souvent d'avoir le choix entre l'initialisation à 0 ou à 1 d'un compteur...

2 Le premier message comporte l'indication `1ème` ; pour obtenir `1er`, il faudrait introduire, dans la boucle, un choix entre deux affichages, basé sur la condition `i = 1`.

Exercice 5.5 Modifier le programme de doublement de capital du paragraphe 1.3.2, de manière qu'il affiche, outre le capital obtenu chaque année, un numéro d'année, comme suit :

```
donnez le capital à placer et le taux : 10000 0.12
capital, à l'année 1 :     11200.00
capital, à l'année 2 :     12544.00
   .....
capital, à l'année 6 :     19738.23
capital, à l'année 7 :     22106.82
```

Exerice 5.6 Écrire un programme qui lit une suite de caractères, terminée par un point, et qui affiche le nombre de caractères lus (point non compris).

3.3 Imposer un nombre de tours

Nos précédents exemples utilisaient un compteur de répétition de façon passive. Mais il est facile d'exploiter « activement » le compteur pour imposer un nombre de répétitions et, donc, pour réaliser une structure de boucle inconditionnelle, en utilisant l'une des structures de répétion conditionnelle déjà présentées. En voici quelques exemples.

3.3.1 Exemple 1

Voici tout d'abord un programme qui affiche, au fur et à mesure, les carrés de 3 valeurs entières fournies par l'utilisateur. Il ressemble à celui du paragraphe 1.1, mais, cette fois, le nombre de valeurs à traiter (ici, 3) est imposé dans le programme.

```
entier n                  // pour le nombre fourni par l'utilisateur
entier i                  // compteur du nombre de valeurs traitées
i := 1
tant que i <= 3 répéter   // attention à la condition : i<=3
  { écrire «donnez un nombre entier : »
    lire n
    écrire «voici son carré : », n*n
    i := i + 1
  }
```

```
donnez un nombre entier : 4
voici son carré : 16
donnez un nombre entier : 12
voici son carré : 144
donnez un nombre entier : 8
voici son carré : 64
```

Affichage des carrés de 3 valeurs fournies en données (1)

3.3.2 Exemple 2

L'exemple précédent utilisait une répétition `tant que`. Nous aurions pu également utiliser une répétition `jusqu'à` :

```
entier n                  // pour le nombre fourni par l'utilisateur
entier i                  // compteur du nombre de valeurs traitées
i := 0
répéter
 { écrire «donnez un nombre entier : »
   lire n
   écrire «voici son carré : », n*n
   i := i + 1
  }
jusqu'à  i >= 3           // condition d'arrêt
```

Affichage des carrés de 3 valeurs fournies en donnés (2)

Remarques

1 Attention à ne pas utiliser `i > 3` comme condition d'arrêt de la boucle ; on traiterait alors 4 valeurs !

2 Nous avons utilisé une condition d'inégalité comme condition de poursuite. La logique veut que l'on puisse tout aussi bien utiliser `i=3` au lieu de `i>=3`. Cependant, la première risque, en cas d'erreur de programmation, de conduire à une boucle infinie si, par malchance, l'égalité n'est jamais vraie. C'est pourquoi, on préfère généralement utiliser la condition d'arrêt la plus large.

3 Il serait possible ici d'initialiser différemment notre compteur de boucle, en modifiant la condition d'arrêt, par exemple :

```
i := 1
répéter
     .....
jusqu'à i >= 4        // ou encore i > 3 ou encore i = 4
```

ou même, de façon totalement artificielle :

```
i = 3
répéter
     .....
jusqu'à i >= 6         // ou encore i >5 ou encore i = 6
```

3.3.3 Exemple 3

Voici une adaptation du premier exemple, de façon qu'il puisse traiter un nombre quelconque de valeurs, fourni préalablement par l'utilisateur. Nous avons reproduit deux exemples d'exécution : dans le second, l'utilisateur demande 0 valeur à traiter.

```
entier nv                 //nombre de valeurs à traiter
entier n                  // pour le nombre fourni par l'utilisateur
entier i                  // compteur du nombre de valeurs traitées
écrire «combien de valeurs à traiter :»
lire nv
i := 1
tant que i <= nv répéter
  { écrire «donnez un nombre entier :»
    lire n
    écrire «voici son carré : », n*n
    i := i + 1
  }
écrire «fin du programme»
```

```
combien de valeurs à traiter :
2
donnez un nombre entier :
5
voici son carré : 25
donnez un nombre entier :
11
voici son carré : 121
fin du programme
```

```
combien de valeurs à traiter
0
fin du programme
```

*Affichage des carrés d'un nombre donné de valeurs fournies en données (*tant que*)*

Remarques

1 Dans les deux premiers exemples, le nombre de tours de boucle est connu lors de l'écriture du programme ; il n'en va plus de même dans le troisième exemple où il n'est connu qu'au moment de l'exécution (et il peut différer d'une exécution à la suivante).

2 Dans le dernier exemple, la répétition `tant que` est mieux adaptée que la répétition `jusqu'à`, dans la mesure où elle permet de prendre facilement en compte le cas où l'utilisateur fournit 0 (ou même un nombre négatif) comme nombre de valeurs à traiter ; en effet, dans ce cas, on obtient aucun tour de boucle, tandis qu'avec une répétition `jusqu'à`, on en obtiendrait quand même un.

3 Nous avons présenté la notion de compteur en vue de réaliser des boucles inconditionnelles ; mais il existe beaucoup d'autres circonstances dans lesquelles un compteur est utile, comme nous aurons l'occasion de le voir.

4 La répétition inconditionnelle

4.1 Exemples d'introduction

Dans la plupart des langages, lorsque l'on exploite un compteur de façon active pour imposer le nombre de tours d'une répétition `tant que`, il est possible de simplifier les choses en faisant appel à une instruction de répétition inconditionnelle particulière.

4.1.1 Exemple 1

Reprenons l'exemple du paragraphe 3.3.1, dans lequel apparaissait ce canevas :

```
i := 1                    // initialisation du compteur à 1
tant que i <= 3           // condition de poursuite : i <= 3
  { ......                // instructions à répéter
    i := i + 1            // +1 sur le compteur
  }
```

Dans la plupart des langages, on peut regrouper dans une même instruction, le nom du compteur, sa valeur initiale et sa valeur finale, d'une manière voisine de ceci :

```
répéter pour i := 1 à 3
  { ......                // instructions à répéter
  }
```

Voici ce que deviendrait notre programme complet, utilisant cette nouvelle structure :

```
entier n           // pour le nombre fourni par l'utilisateur
entier i           // compteur du nombre de valeurs traitées
répéter pour  i := 1 à 3     // pour répéter 3 fois le bloc qui suit
  { écrire «donnez un nombre entier :»
    lire n
    écrire «voici son carré : », n*n
 }
```

Affichage des carrés de 3 valeurs fournies en données

Remarque

Ici, encore, nous obtiendrions le même résultat en remplaçant notre instruction `répéter pour` par l'une des suivantes (seul le nombre de tours ayant de l'importance ici, la valeur du compteur n'étant pas utilisée en tant que telle) :

```
répéter pour i := 0 à 2
répéter pour i := 10 à 12
répéter pour i := -2 à 0
```

4.1.2 Exemple 2

Dans notre exemple précédent, le nombre de tours était parfaitement déterminé. Il était fixé par la valeur des deux constantes qui suivaient le mot `pour` et qui fixaient la valeur initiale et la valeur finale du compteur. Par exemple, avec :

```
répéter pour i := 1 à 25
```

le programme effectuait 25 répétitions.

En fait, dans tous les langages, ces valeurs initiales et finales du compteur peuvent être contenues dans une variable et, même, plus généralement être fournies sous forme d'une expression de type `entier`. Ceci est possible car ce n'est qu'à l'exécution que l'on a besoin de connaître le nombre de tours à réaliser effectivement (le traducteur du programme n'a, quant à lui, pas besoin de cette information).

Voici comment nous pourrions réécrire notre exemple du paragraphe 3.3.1 :

```
entier nv                 // nombre de valeurs à traiter
entier n                  // pour le nombre fourni par l'utilisateur
entier i                  // compteur du nombre de valeurs traitées
écrire «combien de valeurs à traiter :»
lire nv
répéter pour i := 1 à nv
  { écrire «donnez un nombre entier :»
    lire n
    écrire «voici son carré : », n * n
  }
écrire «fin du programme»
```

Affichage des carrés d'un nombre donné de valeurs fournies en données (boucle pour*) (1)*

4.2 Conventions d'écriture

D'une manière générale, nous conviendrons qu'une répétition inconditionnelle se note ainsi :

```
répéter pour compteur := début à fin
    instruction
```

La répétition pour

- *compteur* : variable de type entier ;
- *début*, *fin* : expressions de type entier ;
- *instruction* : instruction de base, bloc d'instructions ou instructions structurées (choix ou boucle).

Remarques

1 Lorsque le problème s'y prête, cette nouvelle instruction de répétition est plus facile à employer que les canevas correspondants utilisant une répétion `tant que`. Elle est plus brève et il est plus facile de déterminer le nombre de répétitions effectuées, ainsi que la valeur attribuée au compteur à chaque tour de boucle.

2 La variable utilisée comme compteur doit avoir été déclarée : en revanche, elle n'a pas besoin d'être initialisée puisque ce sera fait par l'instruction de répétition elle-même.

3 L'instruction `répéter pour` indique, non seulement un nombre de tours de boucle, mais elle fournit aussi le nom du compteur et les valeurs limites. Or, dans certains cas, on peut avoir besoin de répéter des instructions un certain nombre de fois, sans avoir besoin d'exploiter le compteur, ni de fixer des valeurs limites. Une telle instruction pourrait se noter, par exemple :

```
répéter 6 fois
```

Nous disposerions là d'une structure de répétition inconditionnelle supplémentaire qui ne serait qu'un cas particulier de la précédente. Peu de langages possèdent une telle structure de simple répétition et nous n'en introduirons donc pas ici.

Exercice 5.7 Écrire les carrés des nombres entiers de 7 à 20.

Exercice 5.8 Lire deux nombres entiers dans les variables `nd` et `nf` et écrire les doubles des nombres compris entre ces deux limites (incluses).

4.3 Utiliser le compteur dans une répétition inconditionnelle

Dans une répétition inconditionnelle, la variable citée comme compteur est une variable comme n'importe quelle autre. On peut donc en utiliser la valeur dans la répétition. Voici comment nous pourrions réécrire le programme du paragraphe 4.1.2 en affichant le numéro du nombre à lire (cette fois, nous avons bien pris soin d'afficher `1er` et non `1ème`) :

```
entier nv                  // nombre de valeurs à traiter
entier n                   // pour le nombre fourni par l'utilisateur
entier i                   // compteur du nombre de valeurs traitées
écrire «combien de valeurs à traiter :»
lire nv
répéter pour i := 1 à nv
  { si i = 1 alors écrire «donnez un 1er nombre entier :»
             sinon écrire «donnez un », i, «ème nombre entier :»
    lire n
    écrire «voici son carré : », n*n
  }
écrire «fin du programme»
```

```
combien de valeurs à traiter :
2
donnez un 1er nombre entier :
40
voici son carré : 1600
donnez un 2ème nombre entier :
6
voici son carré : 36
fin du programme
```

Affichage des carrés d'un nombre donné de valeurs fournies en données (boucle pour*) (2)*

4.4 Éviter d'agir sur le compteur dans la boucle

Lorsque dans un programme, vous rencontrez une instruction telle que :

```
répéter pour i := 1 à 20
```

vous pouvez raisonnablement vous attendre à ce que les instructions concernées soient effectivement répétées 20 fois. Ce sera généralement le cas. Toutefois, certaines maladresses pourraient compromettre cela. Nous vous proposons d'examiner les plus courantes.

Si vous écrivez :

```
répéter pour i := 1 à 20
   { lire a
     i := i - 1
     écrire a
   }
```

le compteur `i` diminue de un dans la boucle et augmente de un à la fin de chaque tour. Dans ces conditions, sa valeur ne dépasse jamais un et la boucle ne se termine jamais.

D'une manière générale, si vous modifiez la valeur du compteur à l'intérieur de la boucle, vous en pertubez le déroulement. Il est vraisemblable que vous n'effectuerez pas le nombre de tours souhaités.

Dans certains cas, il se peut que vous trouviez astucieux de modifier cette valeur du compteur. Supposons que vous ayez à lire et à réécrire des valeurs entières lues au clavier, jusqu'à ce ce que vous en ayez trouvé 20 positives. Vouz pourriez songer à employer une boucle avec compteur, de cette façon :

```
répéter pour i := 1 à 20
  { lire a
    si a <=0 alors i := i - 1
    écrire a
  }
```

Certes, cela fonctionne ! Mais ne trouvez-vous qu'on trompe le lecteur du programme en faisant croire qu'il s'agit d'instructions répétées 20 fois seulement. En effet, en toute rigueur, le problème posé correspond plus à une répétion conditionnelle : on traite des valeurs jusqu'à ce que l'on en ait trouvé 20 positives. La formulation suivante serait alors bien mieux adaptée :

```
i := 0
répéter
  { lire a
    si a >0 alors i := i + 1
    écrire a
  }
jusqu'à i >= 20
```

D'une manière générale, dans une boucle avec compteur, il est fortement conseillé de ne modifier à l'intérieur de la boucle, ni la valeur du compteur, ni les valeurs limites lorsque celles-ci dépendent des variables. Ce genre de pratiques est formellement interdit dans certains langages, tout en restant souvent accepté des traducteurs (nous verrons, dans la rubrique Côté langages que parfois, une telle vérification est même impossible).

4.5 Compteur et boucle pour

L'exemple précédent montre bien que le fait d'avoir besoin d'un compteur dans une boucle n'implique pas nécessairement l'utilisation d'une répétition `pour`. Voici un autre exemple : il s'agit d'une adaptation du programme du paragraphe 1.3.1 qui affichait les voyelles d'un mot, dans lequel nous limitons à 30 le nombre de caractères fournis par l'utilisateur. La condition d'arrêt devient alors : caractère espace rencontré ou 30 caractères fournis. Notez que pour faciliter les modifications éventuelles de notre programme, nous avons utilisé une « constante symbolique » `nCarMax` pour le nombre maximal de caractères.

```
entier constant nbCarMax := 30      // nombre maximum de caractères lus
caractère c                         // caractère courant
entier i                            // compteur de caractères lus
i := 0
écrire «donnez un mot écrit en minuscules, terminé par un espace»
répéter
   lire c
   si c='a' ou c='e' ou c='i' ou c='o' ou c='u' ou c='y'
        alors écrire c
   i := i + 1
jusqu'à c = ' ' ou i >= nbCarMax
```

```
donnez un mot, écrit en minuscules, terminé par un espace :
anticonstitutionnellement
aioiuioeee
```

Affichage des voyelles d'une suite d'au maximum 30 caractères

4.6 Un tour pour rien

Nous avons vu que nous pouvions utiliser une instruction telle que :

```
répéter pour i := 1 à n
```

ou encore :

```
répéter pour i := n à p
```

Que se passera-t-il dans le premier cas si `n` vaut 0. Que se passera-t-il dans le deuxième cas si la valeur de `p` est inférieure à celle de `n` ? En fait, nous avons défini la répétition `pour` à partir d'une répétition `tant que`. Ce qui signifie que dans les situations évoquées, on ne fera aucun tour de boucle. Cependant, nous n'avons pas la garantie que tous les langages procéderont ainsi. En particulier, certains langages (de plus en plus rares !) pourront utiliser une formulation `jusqu'à` et considérer que :

```
répéter pour i := n à p
  { .....    // instructions à répéter
  }
```

est équivalent à :

```
i := n
répéter
  { .....   // instructions à répéter
    i := i + 1
  }
jusqu'à i > p
```

Dans ce cas, vous voyez que avec `n=5` et `p=2`, nous obtiendrions quand même un tour de boucle, avec i valant 5.

4.7 Le compteur en dehors de la boucle

Considérons cette situation :

```
entier i
   .....    // que vaut i ici ?
répéter pour i := 1 à 5
  {   ..
  }
   .....    // et ici ?
```

Il va de soi que la valeur de `i` n'est pas définie avant d'entrer dans la boucle, ce qui n'a rien de bien gênant. En revanche, les choses sont moins claires en ce qui concerne la valeur de `i` après la sortie de la boucle. Certains langages considéreront que celle-ci n'est pas définie. En pratique, elle aura souvent une valeur, celle qui aura mis fin à la boucle, c'est-à-dire ici 6 si le langage a finalement utilisé une répétition `tant que` et 5 s'il a utilisé une répétition `jusqu'à`. Quoi qu'il en soit, il n'est pas raisonnable de chercher à exploiter cette valeur.

Côté langages

Les répétitions tant que et jusqu'à

Les langages C, C++, Java, C# et PHP définissent ces deux structures avec la même syntaxe. La répétition `tant que` se présente ainsi :

```
while (condition)
{ .....
}                    // attention : pas de point-virgule ici
```

La répétition `jusqu'à` se présente ainsi :

```
do
{ .....
}
while (condition) ;  // attention au point-virgule
                     // et à la condition de «poursuite»
```

On notera qu'ici, la condition régissant la boucle est une condition de poursuite ; il s'agira donc de la condition contraire de la condition d'arrêt que nous utilisons dans la répétition `jusqu'à`.

Python, en revanche, ne dispose que de la répétition `tant que` qui se présente ainsi

```
while condition :
   .....     # aligner les instructions du «bloc»
   .....
```

Dans tous ces langages, la `condition` est tout naturellement une expression booléenne. Mais, en C/C++, elle peut aussi être une expression arithmétique quelconque (caractère, entière ou réelle) ; dans ce cas, toute valeur non nulle est traitée comme « vrai » et seule la valeur nulle est traitée comme « faux ».

La répétition pour

Python dispose d'une instrution traduisant presque littéralement notre instruction répéter puisqu'elle se présente ainsi (attention, la limite supérieure n'est pas atteinte) :

```
for i in range(1,4) :    # correspond à répéter pour i := 1 à 3 (attention 3 et non 4)
  .....                  # le bloc
  .....                  # suivant
```

En C, C++, C#, Java ou PHP, la répétition incondionnelle est quelque peu atypique comme le montre cet exemple (en PHP, il faudrait utiliser des noms de variable commençant par $) :

```
for (i=0, k=3 ; i <=5 ; i++ , k = k*2)
  { .....
  }
```

Cette instruction est en fait équivalente à une répétition de type `tant que` :

```
i = 0 ;
k = 3 ;
while (i <= 5)
{ .....
  i = i + 1 ;
  k = k * 2 ;
}
```

Bien entendu, « qui peut le plus peut le moins » et cette répétition permet de réaliser une simple boucle avec compteur, comme :

```
for (i = 1 ; i <= 4 ; i++)
{ ..... // ici, i prendra successivement les valeurs 1, 2, 3 et 4
}
```

Les instructions sont bien répétées en donnant à `i` les valeurs allant de 1 à 4. On trouve bien l'équivalent d'une répétion `pour i := 1 à 4`.

On notera que, de par sa nature même, cette instruction n'interdit nullement la modification du compteur à l'intérieur de la boucle, puisque celui-ci n'est pas « désigné » de façon explicite

Exemples langages

Voici comment pourrait se traduire l'exemple du paragraphe 4.5 en C, C++, C#, PHP et Python. Notez que, tout en tenant compte des particularités de ces différents langages, nous avons cherché à nous écarter le moins possible du schéma algorithmique initial. Ainsi, la limitation à 30 caractères est conservée, même dans des langages où l'on pourrait connaître d'emblée le nombre de caractères introduits et, partant, simplifier le code (voire le sécuriser).

C

```
#include <stdio.h>
int main()
{ const int nbCarMax = 30 ;     /* nombre maximum de caractères lus */
  char c ;                      /* caractère courant                */
  int i ;                       /* compteur de caractères lus       */
  i = 0 ;
  printf ("donnez un mot écrit en minuscules, terminé par un espace\n") ;
  do
   { c = getchar () ;    /* la fonction getchar lit un seul caractère */
     if ( c=='a' || c=='e' || c=='i' || c=='o' || c=='u' || c=='y') putchar (c) ;
     i++ ;
   }
  while ( ( c != ' ') && (i < nbCarMax) ) ;
}
```

```
donnez un mot écrit en minuscules, terminé par un espace
anticonstitutionnellement
aioiuioeee
```

En C, lorsqu'il s'agit de lire un seul caractère à la fois, on peut utiliser la fonction `getchar`, au lieu de `scanf`. De même, pour écrire un seul caractère à la fois, on peut utiliser la fonction `putchar` au lieu de `printf`.

C++

```
#include <iostream>
using namespace std ;
int main()
{ const int nbCarMax = 30 ;     // nombre maximum de caractères lus
  char c ;                      // caractère courant
  int i ;                       // compteur de caractères lus
  i = 0 ;
  cout << "donnez un mot écrit en minuscules, terminé par un espace\n" ;
```

```
  do
   { cin >> noskipws >> c ;     // noskipws pour éviter de "sauter les espaces"
     if ( c=='a' || c=='e' || c=='i' || c=='o' || c=='u' || c=='y') cout << c ;
     i++ ;
   }
  while ( ( c != ' ') && (i < nbCarMax) ) ;
}
```

```
donnez un mot écrit en minuscules, terminé par un espace
anticonstitutionnellement
aioiuioeee
```

Notez que, par défaut, les lectures au clavier ignorent les caractères « espace », y compris dans la lecture d'un caractère. Il faut modifier ce comportement en utilisant `noskipws` avant la lecture d'un caractère.

C#

```
using System;
class Voyelles
{ static void Main()
  { const int nbCarMax = 30 ;     // nombre maximum de caractères lus
    char c ;                      // caractère courant
    int i ;                       // compteur de caractères lus
    i = 0 ;
    System.Console.WriteLine
        ("donnez un mot écrit en minuscules, terminé par un espace") ;
    String ligne = Console.ReadLine() ;   // on lit une "ligne" de texte
    do
    {  c = ligne[i] ;   // c contient le caractère de rang i de la chaîne ligne
       if ( c=='a' || c=='e' || c=='i' || c=='o' || c=='u' || c=='y')
              System.Console.Write (c);
       i++ ;
    }
    while ( ( c != ' ') && (i < nbCarMax) ) ;
  }
}
```

```
donnez un mot écrit en minuscules, terminé par un espace
anticonstitutionnellement
aioiuioeee
```

Ici, nous avons lu l'ensemble des caractères dans une « chaîne de caractères » nommée `ligne`. Ainsi, `ligne[i]` représente le `i+1` ème caractère (le premier étant `ligne[0]`. Nous n'avons pas modifié la condition de terminaison du mot (espace), ce qui signifie que si l'utilisateur oublie cet espace, on risque fort de « déborder » de la chaîne lue. Une démarche plus sûre aurait alors consisté à utiliser la longueur effective de la chaîne lue (`ligne.Length`) pour connaître le nombre de caractères fournis.

Python

Comme avec C#, il nous faut lire préalablement l'ensemble des caractères dans une chaîne nommée ici `ligne` et `ligne[i]` représente le `i+1`ème caractère (le premier étant `ligne[0]`). Par ailleurs, Python ne dispose pas de répétition « tant que ». Nous aurions pu appliquer la transcription en une répétition jusqu'à, telle qu'elle a été présentée au paragraphe 2.3, mais cela aurait conduit à dupliquer le contenu de la boucle. Pour simplifier les choses, nous avons utilisé un indicateur booléen nommé `fini`, initialisé à `False` et qui prend la valeur `True` lorsque un espace a été rencontré ou que 30 caratères ont été considérés. On notera toutefois que cette démarche ne convient pas si l'utilisateur fournit une réponse vide.

```
nbCarMax = 30   # nombre maximum de caractères considérés
print ("donnez un mot écrit en minuscules, terminé par un espace :")
mot =  input()
fini = False
i = 0
while not fini :
   c = mot[i]   # on prend le ième caractère (le premier est de rang 0)
   if  c=="a" or c=="e" or c=="i" or c=="o" or c=="u" or c=="y" :
      print (c)
   if c==" " or i >= nbCarMax-1 :
      fini = True
   i = i + 1
```

```
donnez un mot écrit en minuscules, terminé par un espace :
anticonstitutionnellement
aioiuioeee
```

Là encore, nous pourrions utiliser directement la longueur de la chaîne lue (len(ligne)) pour connaître le nombre de caractères fournis.

PHP

Comme nous l'avons déjà dit, PHP ne dispose pas de lecture en « mode console », les informations étant généralement lues à partir d'un formulaire. Ici, encore, nous avons fixé le mot concerné dans une variable de type chaîne (notez que PHP ne dispose pas du type caractère mais que, même si cela avait été le cas, nous n'aurions pas pu l'utiliser ici puisqu'il nous faut fournir en bloc tous les caractères du mot). L'accès à un caractère donné de la chaîne `$mot` se fait sous la forme `$mot[i]` où i désigne son rang (attention, le premier caractère correspond au rang 0).

```
<?php
 $nbCarMax = 30 ;     // nombre maximum de caractères lus
 $mot = "anticonstitutionnellement " ;    // ne pas oublier l'espace final
 $i = 0 ;
```

```
 do
   { $c = $mot[$i] ;   // on prend le ième caractère (le premier est de rang 0)
     if ( $c=='a' || $c=='e' || $c=='i' || $c=='o' || $c=='u' || $c=='y') echo $c ;
     $i++ ;
   }
  while ( ( $c != ' ') & ($i < $nbCarMax) ) ;
?>
```

```
aioiuioeee
```

Notez que, comme en C#, nous aurions pu utiliser la longueur de la chaîne `$mot` pour connaître le nombre de caractères qu'elle contient.

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 134). 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
}
```

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.

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

```
entier méthode valeurAbs
{ 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

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.

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 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 190, 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 190, 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 190, 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 190 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 168, 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 nombre 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
```

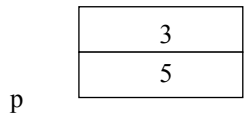
6 Mode des gestion des objets

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
```

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 ces 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.

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.

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

Python est quelque peu atypique. D'une part, dans une méthode, il faut préciser, avec le mot-clé `self`, le paramètre implicite correspondant à l'objet l'ayant appelée. D'autre part, le suffixe `self` doit également apparaître devant les noms des attributs de la classe (qui, par ailleurs ne sont pas déclarés). Par défaut, tous les membres sont publics ; on peut obtenir des membres privés en préfixant leur nom par deux caractères « souligné » (__). Le constructeur se définit par un nom spécifique `__init__` ; il ne peut être surdéfini mais ses paramètres peuvent se voir attribuer des valeurs par défaut.

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
```

En Python, les objets sont également gérés par référence, mais ils sont instanciés par une déclaration de la forme :

```
p = 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, en C++, on peut également 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 langages

Voici comment se présenterait, dans chacun des langages C++, Java, C#, Python et PHP, notre exemple du paragraphe 5.3, page 197, 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
}
```

```
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
```

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
```

Python

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

Comme PHP, Python ne permet de définir qu'un seul constructeur, mais il est possible de prévoir des valeurs par défaut pour les paramètres, comme nous l'avons fait ici pour retrouver nos trois possibilités de construction d'un point.

Notez que les attributs ne sont pas déclarés ; on n'en voit donc pas d'emblée la liste et il est très « facile » d'en créer un supplémentaire par une simple faute d'orthographe !

```
          ## définition de la classe Point
class Point :
  def __init__ (self, x=0, y=0) :
     self.__abs = x ; self.__ord = y
  def deplace (self, dx, dy) :       ## la méthode deplace
     self.__abs += dx ; self.__ord += dy
  def affiche (self) :                ## la méthode affiche
     print ("Je suis un point de coordonnées ", self.__abs, " ", self.__ord)
          ## programme principal
p = Point ()
p.affiche()
q = Point (3)
q.abs=20
q.affiche()
q.__abs=50
q.deplace (3, 6)
q.affiche()
r = Point (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
```

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 `.h`, 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)
 int 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.