2000-2001

UFR Mathématiques de la Décision

Première Année de DEUG Sciences et Technologie mention MASS

Initiation à la programmation Java

Fabrice Rossi

Conditions de distribution et de copie

Table des matières

CHAPITRE 1

Premiers programmes

Sommaire

Introduction

Le but de cet ouvrage est d'enseigner la **programmation** en général, en se basant sur le **langage** `Java`. Il nous faudra d'abord préciser la notion d'ordinateur, même s'il est hors de question, dans le cadre de ce cours, de proposer des explications sur la structure physique des ordinateurs. S'il est en effet important de comprendre qu'un ordinateur est une *machine*, au même titre qu'un magnétoscope, un métier à tisser ou une automobile, sa manipulation pratique reste difficile quand on ne dispose pas d'un *modèle abstrait simple*. L'étude de la réalité physique de l'ordinateur est très intéressante, mais elle est complexe et surtout inutile pour apprendre la programmation.

Pour conduire une automobile par exemple, le conducteur n'a pas besoin de connaître le principe du moteur à explosion. Il doit simplement se représenter mentalement un modèle de la voiture comportant un volant, trois pédales (embrayage, accélérateur et frein) et un levier de vitesse. Il doit aussi connaître d'autres choses importantes pour la sécurité, comme l'utilisation du clignotant, du rétroviseur et des essuie-glaces. Enfin, il doit connaître le code de la route, qui n'est rien d'autre qu'un ensemble de règles logiques et abstraites permettant d'assurer une certaine sécurité.

L'utilisation d'un ordinateur obéit aux mêmes règles générales : nous allons construire un modèle abstrait de l'ordinateur qui sera relativement éloigné de la réalité (il sera surtout très simplifié) et tout notre apprentissage de la "conduite" de l'ordinateur se fera grâce à ce modèle, sans jamais tenir compte de sa réalité physique. Notons que le modèle étudié dans ce cours est lié de façon assez importante au langage `Java`. Cependant, les concepts fondamentaux (variables, algorithme, etc.) sont transposables sans difficulté vers d'autres langages (notamment le C et le C++).

Dans ce chapitre, nous commencerons par présenter un modèle simple de l'ordinateur, puis nous introduirons la notion de **langage** et le principe de la **compilation**. Nous aborderons enfin nos premiers exemples en `Java` en donnant la **forme générale** d'un programme et en précisant quelques règles et conventions d'écriture.

1.1 L'ordinateur abstrait

1.1.1 Introduction

Nous utiliserons dans toute cette présentation la dénomination *ordinateur abstrait* pour désigner le modèle simplifié de l'ordinateur physique que nous proposons. L'ordinateur abstrait se décompose en deux parties (illustrées par la figure 1.1) :

1. le processeur
2. la mémoire

Il s'utilise avec un *programme*.

1.1.2 Le processeur

Pour être rigoureux, il faudrait en fait parler de processeur abstrait, mais cela serait trop fastidieux. Le processeur constitue le pouvoir exécutif de l'ordinateur. C'est en fait la partie centrale de celui-ci. Il peut modifier les informations qui sont stockées dans la mémoire. Il peut aussi permettre l'interaction avec l'utilisateur de l'ordinateur : saisie des choix de celui-ci, affichage à l'écran des résultats des opérations effectuées, etc.

Le processeur comprend un **langage** relativement limité. Ce langage décrit les opérations (les actions) qu'il peut effectuer. Le processeur ne peut rien décider seul, il se contente d'effectuer les opérations indiquées dans le *programme*. Du point de vue pratique, le processeur comprend des instructions binaires, c'est-à-dire des suites de 0 et de 1. Par exemple, `001` peut correspondre à une instruction indiquant au processeur de ne pas considérer l'instruction suivante du programme (il s'agit d'un déplacement de la tête de lecture, cf la section 1.1.4).

1.1.3 La mémoire

L'utilisation principale d'un ordinateur est la manipulation de données. L'ordinateur sert par exemple à faire des calculs scientifiques pour prévoir le temps, pour calculer les contraintes subies par un avion en vol, etc. Plus prosaïquement, l'ordinateur peut vous aider à gérer votre compte en banque, stocker des recettes de cuisine, votre carnet d'adresses, une encyclopédie, etc.

La mémoire de l'ordinateur abstrait est justement l'emplacement dans lequel les informations qu'il traite sont stockées[1]. Les actions que le processeur peut effectuer incluent entre autre la manipulation des données contenues dans la mémoire. Elles comprennent aussi des instructions permettant de placer dans la mémoire des informations saisies par l'utilisateur grâce au clavier de l'ordinateur, ainsi que des instructions permettant d'afficher à l'écran des informations contenues dans la mémoire. Du point de vue pratique, la mémoire stocke des chiffres binaires, c'est-à-dire des 0 et des 1.

1.1.4 Le programme

Dans un premier temps, on peut considérer un programme comme une suite d'ordres à exécuter par le processeur. Dans la pratique, un programme est donc une suite de 0 et de 1, chaque groupe de chiffres devant correspondre à une instruction compréhensible par le processeur.

Comme nous l'avons dit précédemment, le processeur exécute le programme. Pour ce faire, il dispose d'une **tête de lecture** qu'il place d'abord sur la première instruction du programme. Le

[1] Nous parlons ici de la mémoire *vive* (la *RAM*). Les informations sont aussi stockées dans des mémoires permanentes comme les CDROMs, le disque dur, les DVD, etc. La mémoire vive conserve les informations tant qu'elle reste alimentée en courant. Au contraire, les mémoires permanents n'ont pas besoin de consommer de l'énergie pour conserver leurs informations.

processeur exécute alors l'instruction indiquée (cette instruction décrit une action du processeur : par exemple, faire un calcul, en afficher le résultat, etc.). Quand l'instruction a été exécutée, la tête de lecture passe à l'instruction suivante et l'exécute, ainsi de suite jusqu'à la fin du programme.

L'ordinateur abstrait peut exécuter n'importe quel programme, dès lors que celui-ci respecte certaines règles précises.

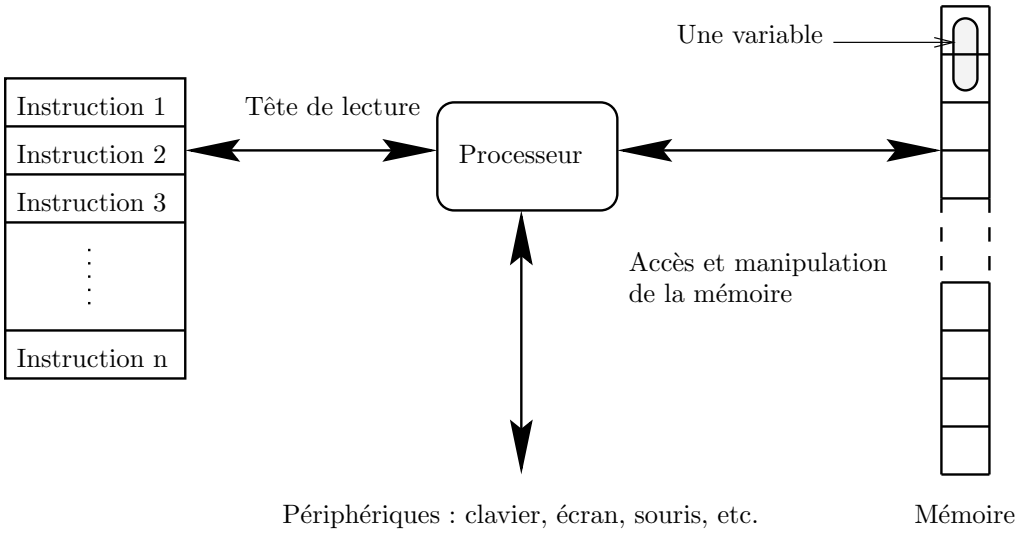

FIG. 1.1 – L'ordinateur abstrait

On peut se demander où est stocké le programme que l'ordinateur exécute. En fait, c'est la mémoire qui le contient. Quand un ordinateur démarre, un dispositif complexe permet au processeur d'aller trouver un programme simple[2] à exécuter. Ce programme se charge de la mise en place des tests et réglages initiaux de la machine, puis place dans la mémoire un autre programme situé sur le disque dur de l'ordinateur[3]. Ce programme est capable de placer dans la mémoire d'autres programmes et de demander au processeur de les exécuter.

1.2 Les langages informatiques

1.2.1 Langage machine et compilateur

Chaque processeur physique[4] ne comprend que des instructions très simples, écrites sous forme de nombres binaires (suite de 0 et de 1). Le codage utilisé est propre au modèle de processeur et est appelé le **langage machine**, pour signifier qu'il s'agit de la langue que parle le processeur.

Il est théoriquement possible d'écrire un programme en langage machine, mais cela reste très long et il ne faut pas se faire d'illusion : les logiciels modernes sont très complexes et il serait en fait matériellement impossible de les écrire en langage machine. De ce fait, on utilise dans la pratique d'autres langages (c'est-à-dire une autre façon de donner des ordres au processeur). Ces langages sont plus facile à lire pour un être humain, mais le processeur n'est pas capable de les comprendre directement. On utilise donc un programme particulier appelé **compilateur**. Le compilateur est capable de traduire un langage évolué en langage machine, acte qu'on appelle la **compilation**.

Dans la pratique, même le compilateur ne travaille pas directement en langage machine. On utilise en effet un **assembleur** qui permet d'écrire en langage machine mais en utilisant des mots

[2] Sur les PC, ce programme est le BIOS.
[3] C'est le système d'exploitation [13], comme par exemple `linux`, cf http://www.linux.org.
[4] Comme par exemple les processeurs PowerPC, Pentium, Athlon, etc.

(en général des abréviations de mots anglais) à la place des nombres binaires. L'assembleur remplace chaque abréviation par le nombre binaire correspondant. Les programmes restent extrêmement difficiles à lire (voir l'exemple de la section 1.3.3), mais sont quand même plus clairs qu'une suite de chiffres binaires. La principale caractéristique qui oppose l'assembleur (le terme désigne à la fois le programme de traduction et l'ensemble des abréviations utilisables) aux autres langages est que l'assembleur est en général spécifique au processeur, alors que les langages évolués ne le sont pas. Pratiquement, cela signifie qu'un programme écrit dans un langage évolué peut être compilé pour différents ordinateurs, avec très peu de modifications (et même aucune en `Java`), alors qu'il faudra en général le réécrire (au moins partiellement) si on avait choisi l'assembleur.

1.2.2 La notion de langage

Définissons maintenant plus clairement ce que nous entendons par langage[5]. Un **langage informatique** est une version très simplifiée d'une langue humaine. Il comporte une orthographe et une grammaire très structurées :

- **L'orthographe** d'un langage (techniquement, c'est le niveau **lexical** du langage) :
  Elle est constituée de deux parties :
  1. une règle pour former des mots valides, les **identificateurs** (cf la section 1.3.1). En français, on ne peut pas considérer "t+)°%," comme un mot (les mots sont en effet constitués d'une suite de lettres). Un langage de programmation définit nécessairement une règle semblable à celle du français ;
  2. une liste de mots clés et de symboles autorisés dans le langage. Nous verrons par exemple que les symboles `=`, `==`, `%`, etc. sont utilisables en `Java`. En français, le mot "kshdfkh" n'a pas de sens. Un langage de programmation définit (comme une langue) un dictionnaire des mots qu'il accepte. Il autorise aussi la définition de l'équivalent des noms propres.
- **La grammaire** d'un langage :
  On l'appelle en fait la **syntaxe** du langage. C'est un ensemble de règles qui déterminent si les symboles du langage sont utilisés correctement. Nous verrons par exemple que `Java` autorise l'écriture suivante : `toto = 2` mais interdit `toto +)=( 2`, alors que les éléments qui interviennent dans le texte sont "orthographiquement" corrects.

Pour finir, un langage informatique possède aussi une **sémantique**. En simplifiant, la sémantique d'un langage est la description de l'effet sur la mémoire et sur le processeur de l'exécution de chacune des instructions. Un langage informatique définit en fait un ordinateur abstrait en précisant (comme nous le verrons pour `Java`) un modèle particulier pour la mémoire et en définissant les instructions que peut comprendre le processeur. Dans la pratique, les modèles abstraits des différents langages restent assez proches, mais il existe tout de même des différences importantes d'un langage à un autre. L'avantage de `Java` est qu'il se base sur un modèle abstrait très évolué : le processeur de `Java` sait faire de nombreuses choses. Pour réaliser dans un langage moins évolué comme le C ce qui est fait grâce à une seule instruction `Java`, on doit parfois utiliser de nombreuses instructions.

1.2.3 Compilation et interprétation

Comme nous l'avons dit précédemment, le texte d'un programme rédigé dans un langage donné est traduit par le compilateur en langage machine. Il existe aussi une autre technique de compilation qui est utilisée pour `Java`. Le principal problème de la compilation telle qu'elle vient d'être décrite est qu'elle dépend de l'ordinateur visé (en particulier du processeur). Si, par exemple, on compile un programme pour un Macintosh[6] le programme résultat (la traduction en langage machine) ne

[5] Le lecteur intéressé pourra se reporter à [1], ouvrage de référence sur la théorie des langages informatiques.
[6] Les Macintosh récents utilisent des processeurs PowerPC.

sera pas utilisable sur un autre ordinateur[7]. De ce fait, quand un vendeur de logiciels souhaite produire un logiciel qui fonctionne sur toutes sortes de machines, il est obligé de prévoir autant de versions qu'il a de types d'ordinateurs. Cette procédure est un peu lourde. L'idée de `Java` est de placer une étape intermédiaire dans la phase de compilation (c'est une méthode relativement nouvelle, même si `Java` n'est pas le précurseur en ce domaine). Au lieu de compiler un programme `Java` afin de fabriquer un programme en langage machine spécifique à un ordinateur particulier, le compilateur `Java` transforme le programme en un autre programme dans le langage machine de la **machine virtuelle Java** (qu'on appelle en général la JVM, pour *Java Virtual Machine*). Ce langage est un langage de bas niveau, beaucoup moins évolué que `Java`. On l'appelle le ***bytecode*** (voir la section 1.3.3 pour un exemple de *bytecode*). Il est plus évolué que les langages machines des processeurs actuels et il est surtout **complètement indépendant de tout ordinateur**. De ce fait, la compilation se fait une fois pour toutes.

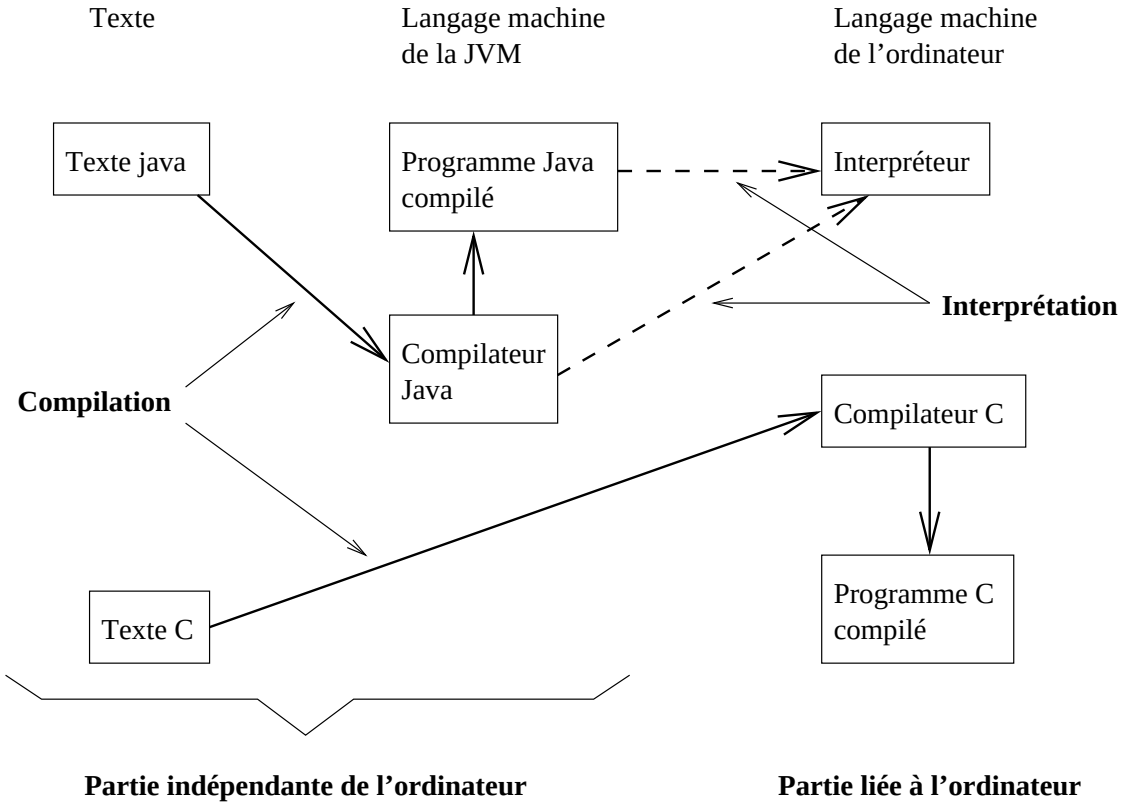

FIG. 1.2 – Compilation et interprétation

Avec un compilateur normal, le programme obtenu est exécuté directement par l'ordinateur. Le processeur réel lit les instructions et les exécute. Avec `Java`, c'est un peu plus compliqué car le programme compilé n'est pas *directement* compréhensible par un ordinateur quelconque. On utilise alors un **interpréteur**, c'est à dire un programme spécial qui traduit *au vol*[8] le programme compilé en instructions pour l'ordinateur. Il y a deux grandes différences entre un compilateur et un interpréteur :

– quand on lance deux fois de suite l'interprétation d'un programme, la traduction est faite deux fois. En d'autres termes, la traduction de l'interpréteur n'est pas conservée alors que celle du compilateur l'est ;

[7] Notamment sur un PC qui utilise un processeur de la famille x86, comme par exemple un Athlon.
[8] C'est-à-dire pendant l'exécution elle-même.

- la somme de la durée de compilation d'un programme et de la durée d'exécution est souvent moins importante que la durée de l'interprétation du même programme : l'interprétation est moins efficace que la compilation.

Pour lancer un programme `Java` compilé, on doit donc utiliser un autre programme alors que pour les langages classiques ce n'est pas le cas. Cette méthode possède des avantages et des inconvénients :

- un interpréteur existe sur à peu près tous les ordinateurs courants, ce qui veut dire qu'un programme en `Java` compilé peut être exécuté sur tout ordinateur, alors que le programmeur ne dispose que d'un ordinateur particulier. Le seul moyen d'atteindre ce but avec un autre langage est de donner le texte du programme afin que chacun puisse le recompiler pour son propre ordinateur. Aucun vendeur de logiciels ne peut se permettre de faire cela[9] ;
- la compilation est plus rapide que pour un langage équivalent car la machine virtuelle est un intermédiaire entre un ordinateur concret et l'ordinateur abstrait ;
- un programme exécuté par interprétation est toujours plus lent que le même programme compilé en langage machine[10].

On peut se demander pourquoi ne pas interpréter directement du `Java`. En fait, ce serait beaucoup trop lent car `Java` est basé sur un ordinateur virtuel très évolué par rapport aux ordinateurs concrets. La traduction directe serait donc très lente. L'idée de la machine virtuelle est d'avoir quelque chose d'intermédiaire : un ordinateur abstrait pour lequel la traduction est suffisamment rapide pour pouvoir faire une interprétation.

REMARQUE

Il est très important de garder à l'esprit que la vie d'un programme comporte deux phases : la compilation et l'exécution. Nous verrons que le compilateur effectue des vérifications assez complexes sur le texte du programme pendant sa traduction. Ces vérifications sont dites **statiques**. Elles évitent une partie des erreurs qui peuvent se produire lors de l'exécution du programme.

Par opposition à la compilation qui constitue la phase **statique** de la vie du programme, on appelle phase **dynamique** l'exécution du programme.

1.3 Forme générale d'un programme `Java`

1.3.1 Les identificateurs

Comme nous l'avons indiqué à la section 1.2.2, un langage de programmation possède une règle de définition des mots valides, une sorte d'orthographe élémentaire régissant en particulier les "noms propres". En `Java`, on peut associer un **identificateur** à certains éléments des programmes. Voici la définition d'un identificateur :

Définition 1.1 *Un **identificateur** est une suite de caractères* autorisés*. Les caractères autorisées sont les lettres de l'alphabet (minuscules et majuscules), le symbole souligné* `_` *et les chiffres. Un identificateur doit commencer par une lettre ou par* `_`*.*

Il est donc interdit de mettre dans un identificateur des symboles spéciaux comme `@`, `+`, etc, ou même des espaces. Le "mot" `1truc` n'est donc pas un identificateur correct par exemple.

[9] Pour être précis, aucun vendeur *classique* ne peut se permettre de faire cela. Ceci étant, des sociétés comme RedHat vendent des logiciels avec leur texte et ne s'en portent pas plus mal. Voir par exemple http://www.redhat.com/.

[10] C'est pourquoi les machines virtuelles utilisent maintenant une technique plus évoluée que l'interprétation simple, qui consiste (en simplifiant) à conserver le résultat de la traduction, ce qui accélère notablement la plupart des programmes. La technique employée s'appelle la compilation *Just In Time* (JIT). On obtient ainsi des programmes presque aussi rapides que par compilation directe.

REMARQUE

Il est important de noter que les majuscules ne sont pas confondues avec les minuscules. Ainsi l'identificateur `compteur` est-il différent de `Compteur`.

1.3.2 Début et fin d'un programme

Comme nous l'avons expliqué dans les sections précédentes, un programme `Java` est donc une suite d'instructions respectant les règles du langage. Pour des raisons complexes que nous expliciterons progressivement dans la suite du cours, il faut ajouter, avant et après les instructions proprement dites, du texte qui permet au compilateur de bien comprendre ce qu'on souhaite faire. Voici un exemple très simple de programme `Java` complet (les numéros de ligne sont présents dans cet exemple pour simplifier l'analyse du programme. Ils ne font pas partie d'un véritable programme) :

Exemple 1.1 :

PremierEssai

```
1 public class PremierEssai {
2   public static void main(String[] args) {
3     int u;
4     u = 2;
5   }
6 }
```

On remarque plusieurs choses :

- la ligne `1` indique que nous sommes en train d'écrire le programme intitulé `PremierEssai` (qui est donc le nom du programme). Le mot `PremierEssai` peut être remplacé par n'importe quel *identificateur* ;
- l'accolade ouvrante qui termine cette ligne indique que le texte du programme commence après la ligne et se termine à l'accolade fermante qui lui correspond (la dernière ligne du texte) ;
- la ligne `2` est la plus difficile à justifier. Nous ne tenterons pas de le faire à ce niveau du cours (nous reviendrons brièvement sur le sens de cette ligne à la section 3.1.2, puis plus en détails au chapitre 6). Nous remarquerons simplement que l'accolade ouvrante indique que le texte du programme commence *véritablement* après la ligne et se termine à l'accolade fermante qui lui correspond (nous verrons au chapitre 4 qu'on écrit en fait un *bloc*) ;
- le texte du programme est donné dans les lignes qui suivent (lignes `3` et `4` dans cet exemple). On le termine par deux accolades fermantes, situées sur des lignes distinctes. Nous verrons la signification des lignes 3 et 4 au chapitre 2 ;
- on remarque que le début d'une ligne est déplacé vers la gauche après chaque accolade ouvrante et vers la droite après chaque accolade fermante. Il s'agit d'une convention de présentation qui rend les programmes plus simples à lire (voir la section 1.3.4).

La forme générale d'un programme `Java` est donc la suivante :

```
public class nom du programme {
  public static void main(String[] args) {
    texte du programme proprement dit : ligne 1
    ligne 2
    ...
    ligne n
  }
}
```

Comme la partie intéressante d'un programme est constituée avant tout des instructions qui suivent la ligne avec `main`, nous écrirons souvent des morceaux de programme dans le texte qui va suivre, c'est-à-dire que nous omettrons souvent les deux premières lignes du texte et les deux dernières accolades.

REMARQUE

Il est très important de noter ceci : un programme est un texte qui est stocké sur l'ordinateur sous forme d'un fichier (comme toutes les informations que contient l'ordinateur). Le nom de ce fichier est **obligatoirement** celui du programme terminé par `.java`. Par exemple pour notre programme `PremierEssai`, le fichier doit impérativement être appelé `PremierEssai.java`. De plus, le compilateur transforme ce programme en un nouveau programme destiné à la machine virtuelle `Java`. Le nouveau programme est alors stocké dans un fichier portant le nom du programme terminé par `.class`. Dans notre exemple, le fichier compilé porte donc le nom `PremierEssai.class`.

1.3.3 Exemple de *bytecode*

A titre d'exemple, voici la traduction en *bytecode* du programme `PremierEssai`. Comme indiqué à la section 1.2.1, le langage machine est quasi impossible à lire. Le *bytecode* proposé ici est donc la version assembleur :

```
1  public class PremierEssai extends java.lang.Object {
2      public PremierEssai();
3      public static void main(java.lang.String[]);
4  }
5
6  Method PremierEssai()
7     0 aload_0
8     1 invokespecial #1 <Method java.lang.Object()>
9     4 return
10
11 Method void main(java.lang.String[])
12    0 iconst_2
13    1 istore_1
14    2 return
```

Les premières lignes indiquent que le programme `PremierEssai` comporte deux méthodes (nous verrons au chapitre 3 ce qu'il faut entendre par méthode). Seule la méthode `main` nous intéresse car elle correspond au programme proprement dit. Les lignes 3 et 4 du programme `PremierEssai` ont été traduites en trois instructions en *bytecode*, données sur les lignes 12, 13 et 14. On voit qu'il y a peu de rapport entre la version `Java` et la version *bytecode*.

1.3.4 Conventions de présentation des programmes

Les règles d'écriture d'un programme correct en `Java` sont très strictes. Cependant, il reste une certaine liberté qu'on souhaite réduire afin de faciliter la lecture des programmes. C'est le but des **conventions**. Il faut bien comprendre que les conventions sont des règles *librement acceptées* par le programmeur. Le compilateur ne s'occupe en aucun cas des conventions et un programme qui ne les respecte pas peut très bien être parfaitement correct (d'ailleurs, un programme qui respecte les conventions peut aussi être incorrect). Dans tout cet ouvrage, nous utilisons les conventions proposées par la société SUN, inventeur du langage `Java` (voir le document [11]).

Les noms de programmes

Nous observerons les conventions suivantes pour les noms de programmes :

1. le nom d'un programme comportant un seul mot (français ou anglais) est écrit entièrement en minuscules, excepté la première lettre. Par exemple on écrit `Bonjour` plutôt que `bonjour` ;
2. le nom d'un programme comportant plusieurs mots est écrit de la façon suivante : la première lettre de chaque mot est en majuscule, toutes les autres lettres étant en minuscules. Les différentes mots sont collés les uns aux autres. Par exemple on écrira `CalculDeLaMoyenne` plutôt que `calcul_de_la_moyenne` ou toute autre solution.

La présentation des programmes

Nous observerons les conventions suivantes pour la présentation des programmes :

1. un programme ne comporte qu'une seule instruction par ligne ;
2. une accolade ouvrante est toujours placée en fin de ligne ;
3. toutes les lignes comprises entre une accolade ouvrante et l'accolade fermante qui lui correspond sont indentées d'un cran[11] vers la droite ;
4. une accolade fermante est seule sur sa ligne et est alignée avec le début de la ligne contenant l'accolade ouvrante qui lui correspond.

REMARQUE

Il est important de noter que pour le compilateur, il n'y a pas de différence entre un espace, une tabulation ou un passage à la ligne. Pour lui, les différents symboles sont équivalents à un simple espace. On peut donc écrire le programme suivant :

PremierEssaiBlanc

```
1  public
2   class
3    PremierEssaiBlanc
4     {
5      public
6       static
7        void
8         main(String[] args)
9          {
10         int u;
11         u = 2;
12   }
13 }
```

Ce programme est parfaitement correct. Il ne respecte pas les conventions de présentation et pour un programmeur entraîné, il est beaucoup moins facile à lire que la version qui respecte les conventions.

1.3.5 Les commentaires

Afin de rendre plus clair le texte d'un programme, on peut y ajouter des *commentaires*. Un commentaire est un texte en français écrit de telle sorte que le compilateur l'ignore.

[11] En d'autres termes, on ajoute des blancs au début de la ligne.

Exemple 1.2 :

Voici un exemple de programme utilisant des commentaires :

Commentaires

```
1 // Voici un premier commentaire
2 public class Commentaires {
3   public static void main(String[] args) {
4     /* On doit donner dans les lignes suivantes
5        les instructions qui constituent le programme */
6   }
7 }
```

Nous avons ici les deux types de commentaires :

1. toute ligne qui débute par deux *slash* (`//`) est une ligne de commentaire, c'est-à-dire que compilateur (et donc le processeur) ne tient pas compte de ce qu'elle contient ;
2. quand on écrit un *slash* suivi d'une étoile (`/*`), on débute un commentaire. On peut alors écrire tout le texte qu'on souhaite, en passant à la ligne, etc. La fin du commentaire est indiquée grâce à une étoile suivie d'un *slash* (`*/`). Ceci signifie qu'un commentaire ne peut pas contenir une telle séquence[12].

1.4 Conseils d'apprentissage

Certains éléments du présent chapitre sont avant tout destinés à replacer l'apprentissage de `Java` dans un cadre plus général, celui de la programmation. Du point de vue pratique, les éléments les plus importants sont les suivants :

- La distinction entre les deux phases de la vie d'un programme est centrale et nécessaire à une bonne compréhension des mécanismes du langage. Il faut donc bien retenir l'opposition **compilation** *versus* **exécution**, qui correspond à l'opposition **statique** *versus* **dynamique**.
- La notion d'**identificateur** est centrale à tous les langages de programmation. Il est donc important de bien retenir sa définition est `Java`.
- La **forme générale** d'un programme doit impérativement être connue parfaitement car le compilateur ne tolère aucune erreur.
- Les **conventions de présentation** sont une aide précieuse dans l'écriture et surtout dans la relecture des programmes.

[12] Ce qui n'est pas dramatique !

CHAPITRE 2

Variables et calculs

Sommaire

Introduction

Dans le chapitre précédent, nous avons étudié un modèle abstrait simple de l'ordinateur. Ce modèle de bas niveau convient pour une description générale des ordinateurs. Dans la pratique, chaque langage propose son propre modèle abstrait, décrivant en particulier l'organisation de la mémoire et les instructions compréhensibles par le processeur. Dans le présent chapitre, nous allons commencer l'étude de l'ordinateur abstrait spécifique au langage `Java`.

Nous consacrerons plus particulièrement ce chapitre à la **mémoire**. En effet, elle stocke les informations que le processeur manipule. Sans stockage, on ne peut pas écrire de programme. Nous verrons donc comment la mémoire est organisée pour `Java` et nous étudierons la notion de **variable**. Les variables permettent à un programme d'imposer une organisation adaptée de la mémoire, ainsi qu'une interprétation de son contenu (nombres entiers, nombres réels, texte, etc.). Le langage utilisé dans un programme détermine d'ailleurs les interprétations possibles par l'intermédiaire de la notion de **type**.

Après avoir compris le modèle de la mémoire en `Java`, nous présenterons deux instructions très importantes, briques élémentaires de tout programme. La **déclaration de variable** permet à un programme de préciser les variables qu'il souhaite utiliser, c'est-à-dire l'utilisation de la mémoire qu'il envisage. L'**affectation** permet de placer dans la mémoire les informations à manipuler. Nous verrons en particulier comment faire faire des calculs au processeur.

2.1 Mémoire et variables

2.1.1 Modèle de la mémoire en Java

Nous avons vu dans le chapitre précédent que la mémoire de l'ordinateur est capable de stocker des chiffres binaires, c'est-à-dire des 0 et des 1. En Java, la mémoire est de plus organisée. On

considère qu'elle est constituée d'un ensemble de cases identiques. Chaque case permet de stocker un *octet*[1] c'est-à-dire une suite de 8 chiffres binaires, aussi appelés bits.

De plus, la mémoire n'est pas manipulée directement sous sa forme élémentaire par les programmes `Java`. En effet, comment peut-on faire par exemple pour représenter un texte en français à partir de chiffres binaires (si on veut utiliser l'ordinateur pour écrire une lettre) ? La réponse à cette question est typiquement un détail technique dont l'utilisateur n'a pas besoin de s'occuper. De ce fait, le modèle d'ordinateur abstrait de Java permet de manipuler la mémoire de façon plus évoluée que sous sa forme élémentaire.

2.1.2 Regroupement de cases et notion de type

Il semble évident que pour stocker dans la mémoire une valeur entière comprise par exemple entre 0 et 1000, une seule case ne va pas suffire. Le nombre de cases élémentaires nécessaire pour stocker une information dépend donc de la *nature* de l'information considérée.

L'ordinateur abstrait de `Java` possède la faculté d'associer à chaque information qu'il manipule un *type*, qui indique la nature de cette information. Il peut ainsi manipuler diverses informations de natures différentes comme des nombres entiers, des nombres réels, des caractères, etc. Il sait exactement combien de cases de la mémoire il doit utiliser pour stocker une valeur d'une certaine nature. L'utilisateur n'a pas besoin de savoir *comment* l'ordinateur découpe l'information afin de la répartir dans les différentes cases car ce dernier possède des instructions qui permettent de manipuler les valeurs en question *en tenant compte de leur nature*.

Voici la liste des types directement manipulables en `Java`. Ce sont les **types fondamentaux** :

Nom du type	Description	Cases
`byte`	entier compris entre -128 et 127 ($2^7 - 1$)	1
`short`	entier compris entre -32768 et 32767 ($2^{15} - 1$)	2
`int`	entier compris entre -2147483648 et 2147483647 ($2^{31} - 1$)	4
`long`	entier compris entre $-(2^{63})$ et $2^{63} - 1$	8
`float`	nombre réel en simple précision (environ 7 chiffres significatifs)	4
`double`	nombre réel en double précision (environ 15 chiffres significatifs)	8
`boolean`	la valeur `true` ou `false`	1[2]
`char`	un caractère	2

Nous verrons plus tard que chaque type est associé à un certain nombre d'opérations. Dans un programme, l'utilisation d'une valeur d'un certain type se fera grâce aux instructions possibles pour ce type. On verra ainsi comment additionner deux valeurs entières, etc.

REMARQUE

Dans la pratique, les types les plus importants sont les types `int` et `boolean` car ils interviennent directement dans les structures de contrôle qui permettent d'écrire des programmes intéressants (voir les chapitres 4 et 5).

En général, les calculs numériques sont réalisés grâce au type `double`. Enfin, l'interaction avec l'utilisateur passe par les chaînes de caractères et donc par le type `char`. On peut dire que les autres types sont beaucoup moins utilisés.

A titre illustratif, voici quelques exemples de représentation binaire :

[1] *byte* en anglais.

[2] La situation pour les `boolean`s est en fait très complexe. Quand on considère un *groupe* de `boolean`s, chaque `boolean` occupe effectivement une case. Par contre, un `boolean` considéré de façon isolée occupe 4 cases.

Type	valeur numérique	représentation binaire
`int`	1	00000000000000000000000000000001
`short`	1	0000000000000001
`float`	1	00111111100000000000000000000000
`int`	-1	11111111111111111111111111111111
`short`	-1	1111111111111111
`float`	-1	10111111100000000000000000000000
`int`	5	00000000000000000000000000000101
`short`	5	0000000000000101
`float`	5	01000000101000000000000000000000

On constate que la représentation binaire n'est pas très "parlante"! Fort heureusement, nous n'aurons pas à nous en servir car `Java` permet l'utilisation directe des valeurs numériques usuelles (entières et réelles). Il n'est d'ailleurs pas prévu de pouvoir indiquer simplement dans un programme `Java` la représentation binaire d'une valeur. Excepté pour le cas un peu particulier des `char`s (cf la section 2.5.3), le langage `Java` masque complètement le codage des valeurs utilisées.

2.1.3 Les variables

En `Java`, la mémoire est en fait un ensemble de groupes de cases. Chaque groupe de cases représente une information dont l'ordinateur connaît le type. Le problème est maintenant de trouver un moyen pour l'utilisateur de manipuler ces informations. Pour ce faire, il faut que le programme (plus précisément les instructions du programme) puisse désigner un groupe de cases particulier. Le problème est résolu par la notion de **variable** :

Définition 2.1 *Une **variable** est un groupe de cases de la mémoire de l'ordinateur abstrait qui contient une valeur dont la nature est déterminée par le **type** du groupe de cases (appelé type de la variable). Chaque variable est désignée par un nom, qui doit être un* identificateur *valide.*

Dans une instruction du programme, l'utilisateur peut faire référence à une variable en utilisant son identificateur. Comme l'ordinateur abstrait associe à l'identificateur le type de la variable, on est sûr que l'instruction va interpréter correctement ce groupe (et ne pas prendre un caractère pour un entier par exemple).

Il est important de noter que l'organisation de la mémoire est spécifique à chaque programme. Deux programmes différents utiliseront la mémoire de façon différente, bien que la mémoire abstraite sous-jacente soit la même. De ce fait, chaque programme doit comporter des instructions qui indiquent au processeur comment organiser la mémoire, c'est-à-dire qui décrivent les variables que les autres instructions pourront utiliser.

Il faut aussi noter que l'utilisation des variables est **l'unique** moyen pour le processeur de manipuler la mémoire.

2.1.4 Déclaration de variables

Avant d'utiliser une variable, un programme doit commencer par donner le nom et le type de celle-ci, grâce à une instruction de **déclaration**.

Exemple 2.1 :

Supposons par exemple, que nous souhaitions écrire un programme qui calcule la moyenne de deux nombres entiers. Nous devons utiliser trois variables, une pour chaque nombre entier et une pour le résultat du calcul (la moyenne, qui n'est pas nécessairement entière). Le début du

programme comportera alors les lignes suivantes[3] :

```
int premier ;
int second ;
double moyenne ;
```

La figure 2.1 montre comment on peut représenter l'effet sur la mémoire de la déclaration de variables. Nous sommes bien loin ici de la réalité physique de l'ordinateur. La mémoire est considérée simplement comme l'ensemble des variables déclarée. Le groupe de cases mémoire correspondant à une variable est représenté par une seule case dans laquelle on indique le type de la variable (puis sa valeur, comme on le verra par la suite). La case pour la variable de type `double` est deux fois plus grosse que celles utilisées par les variables de type `int`, afin de rappeler qu'un `double` occupe huit octets, alors qu'un `int` occupe seulement 4 octets (en général, on simplifie la présentation sans faire ce rappel). Enfin, l'identificateur de la variable est indiqué devant celle-ci. Dans la réalité, l'effet de la déclaration d'une variable est beaucoup plus complexe, mais cette représentation est largement suffisante pour pouvoir programmer.

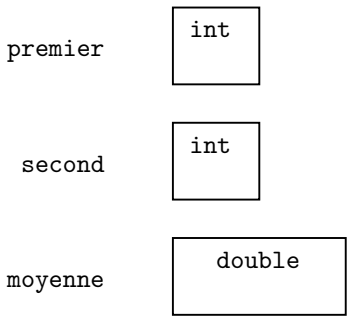

FIG. 2.1 – Déclaration de variables

La déclaration d'une variable consiste simplement à indiquer son type suivi d'un espace puis du nom de la variable (un identificateur) et se termine par un point virgule. De façon générale une déclaration de variable a la forme suivante :

type identificateur ;

Il est possible de simplifier la déclaration des variables qu'on souhaite utiliser dans un programme en regroupant entre elles les déclarations de variables du même type.

Exemple 2.2 :

Reprenons l'exemple 2.1. Dans celui-ci, on déclare deux variables de type `int`. Il est possible de remplacer les deux lignes de déclaration par l'unique ligne suivante :

```
int premier, second ;
```

Le principe d'une **déclaration multiple** est très simple. Au lieu de donner un seul identificateur par ligne, on en donne autant qu'on souhaite, séparés entre eux par des virgules. La signification de la déclaration est simple, une variable du type indiqué sera créé pour chaque identificateur. La forme générale est ainsi :

type identificateur_1, identificateur_2,..., identificateur_n ;

[3] Nous indiquons ici le morceau intéressant du programme. Il ne s'agit pas d'un programme complet.

REMARQUE

L'écriture précédente n'est pas utilisable telle quelle. Il est impossible dans un programme d'utiliser une ellipse pour indiquer qu'on veut un nombre de variable dépendant d'un paramètre (ici `n`). Il s'agit simplement d'une facilité d'écriture pour indiquer qu'un nombre arbitraire d'identificateurs peut être utilisé.

2.1.5 Conventions pour les noms de variables

Pour les noms de variables, nous utiliserons les mêmes conventions que pour les noms de programmes (cf la section 1.3.4), à une exception près : la première lettre du nom d'une variable sera toujours une minuscule. Par exemple on écrira `noteDeMath` plutôt que `note_de_math` ou toute autre solution.

REMARQUE

Les conventions sont très importantes car elles sont suivies par l'ensemble (ou presque) des programmeurs `Java`. De ce fait, elles permettent de relire facilement un programme et surtout de se souvenir plus facilement des noms des variables. Si on écrit un programme dans lequel on manipule une "note de math", la convention précise que seul l'identificateur `noteDeMath` sera utilisé. La mémorisation d'une seule règle permet ensuite de s'intéresser au nom usuel de la variable, sans avoir à attacher d'importance à l'aspect typographique.

2.2 Affectation

2.2.1 Valeurs littérales

Comme nous l'avons souvent répété, l'ordinateur manipule des informations. Il est donc indispensable de pouvoir décrire dans un programme ces informations. Si on veut par exemple se servir de l'ordinateur pour faire des calculs, on doit pouvoir placer dans les variables des valeurs numériques entières ou réelles. De façon générale, il est utile de pouvoir donner dans un programme une valeur correspondant à un type fondamental. L'ordinateur abstrait possède des règles permettant d'écrire des valeurs correspondant à chacun des types fondamentaux donnés dans la section 2.1.2. Une telle valeur s'appelle une *valeur littérale*.

Les valeurs littérales ne sont pas des instructions. Elles seront utilisées à l'intérieur d'instruction comme l'affectation mais ne peuvent en aucun cas apparaître seules.

Valeurs littérales entières

Quand on écrit un entier de façon usuelle, l'ordinateur abstrait le considère comme une valeur littérale de type `int`. On écrira donc dans un programme `123`, `2414891`, etc. En fait, toute suite (par trop longue !) de chiffres est une valeur littérale de type `int`.

Si on souhaite donner une valeur littérale de type `long`, on doit faire suivre l'entier de la lettre `l` ou `L`. Par exemple, `2l` ne désigne pas la valeur entière 2 représentée comme un `int`, mais représentée comme un `long` (on utilise donc 8 octets au lieu de 4).

Il n'existe aucun moyen de donner une valeur littérale de type `byte` ou `short`. L'écriture `2b` ne signifie en aucun cas la valeur entière 2 représentée comme un `byte`. En fait, elle n'est pas acceptée par l'ordinateur. Nous verrons à la section 2.3.5 qu'on peut quand même utiliser les types `byte` et `short` moyennant certaines précautions.

Valeurs littérales réelles

Quand on écrit un nombre réel de façon usuelle (comme sur une calculatrice), l'ordinateur le considère comme une valeur littérale de type `double`. Rappelons que la partie fractionnaire d'un nombre réel est séparée de sa partie entière par un point (notation anglo-saxonne classique). On écrit par exemple `0.001223` ou encore `454.7788`. On peut également utiliser une notation "scientifique" où `1.35454e-5` par exemple désigne 1.35454 10^{-5} représenté sous forme d'un `double` (`e` pouvant être remplacé par `E`).

Pour obtenir une valeur littérale de type `float`, on fait suivre l'écriture du réel par la lettre `f` ou `F`. Par exemple, `2.5f` désigne le réel 2.5 représenté sous forme d'un `float`, c'est-à-dire en utilisant 4 octets, alors que `2.5` désigne le même réel mais sous forme d'un `double`, c'est-à-dire représenté par 8 octets. Notons qu'il est possible de faire suivre un réel par la lettre `d` ou `D`, ce qui indique (de façon redondante) que c'est une valeur littérale de type `double`.

Autres valeurs littérales

Pour le type `boolean` qui représente une valeur de vérité, il existe deux valeurs littérales seulement : `true` et `false` (c'est-à-dire *vrai* et *faux*).

Pour le type `char` qui représente un caractère, on donne une valeur littérale simplement en encadrant le caractère considéré entre deux apostrophes. On écrit ainsi `'a'`, `'2'`, `'@'`, etc. Pour des détails sur les caractères, on pourra se reporter à la section 2.5.3.

Récapitulation

Pour les valeurs numériques, voici une récapitulation des règles appliquées :

Valeur littérale	Type
entière seule	`int`
entière suivie d'un `l` ou d'un `L`	`long`
entière suivie d'un `f` ou d'un `F`	`float`
entière suivie d'un `d` ou d'un `D`	`double`
réelle seule	`double`
réelle suivie d'un `f` ou d'un `F`	`float`
réelle suivie d'un `d` ou d'un `D`	`double`

On remarque que les modificateurs `d` et `f` peuvent s'appliquer aux entiers et les faire considérer comme des `double`s ou des `float`s. Il faut bien garder à l'esprit qu'un entier est un cas particulier de réel !

Exemple 2.3 :

Voici un tableau qui illustre ces règles :

valeur	`2`	`2.0`	`2.0f`	`2.0d`	`2l`	`2f`	`2d`
type	`int`	`double`	`float`	`double`	`long`	`float`	`double`

Il est important de noter que la lettre éventuelle suit directement le nombre. Il ne peut pas y avoir d'espace entre les deux.

REMARQUE

Comme nous l'avons indiqué précédemment, les types `int` et `double` sont les plus importants des types numériques. Or, les règles ci-dessus indiquent que ce sont justement les valeurs littérales les plus simples à écrire. L'emploi des modificateurs (comme `f` par exemple) sera donc peu fréquent.

2.2.2 Affectation d'une valeur littérale à une variable

Après la déclaration, la seconde instruction que nous allons étudier est *l'affectation*. Nous avons jusqu'à présent parlé du rôle général de l'ordinateur comme instrument de manipulation d'informations. Nous savons que ces informations sont stockées dans la mémoire grâce à des variables. Il nous reste maintenant à être capables de placer des informations dans une variable. Pour ce faire on utilise une *instruction d'affectation*.

Exemple 2.4 :

Reprenons l'exemple du calcul de la moyenne de deux nombres que nous supposerons réels cette fois-ci (exemple 2.1). Pour pouvoir faire cette moyenne, il faut donner la valeur des réels en question. Pour ce faire, nous écrivons la partie de programme suivante :

```
double premier, second, moyenne;
premier = 2.3434;
second = -23.4e-1;
```

Le sens de la première ligne est déjà connu, il s'agit de la déclaration des variables que nous souhaitons utiliser. Les dernières lignes s'interprètent de la façon suivante : chaque ligne du programme indique au processeur de placer la valeur littérale *à droite* du signe égal dans la variable dont l'identificateur apparaît *à gauche* du signe égal. Donc après exécution des deux lignes, la variable `premier` contient la valeur 2.3434 alors que la variable `second` contient la valeur -2.34. Nous avons donc stocké dans la mémoire de l'ordinateur les deux valeurs littérales. La figure 2.2 donne une illustration de l'écriture dans la mémoire des valeurs littérales. Il s'agit encore une fois de fournir un support pour la compréhension des programmes, ce n'est pas exactement de cette façon qu'une affectation est réalisée dans la mémoire de l'ordinateur.

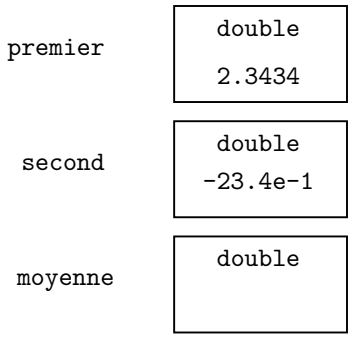

FIG. 2.2 – Affectation

De façon plus générale, l'affectation est l'opération qui consiste à stocker une valeur dans une variable (c'est-à-dire dans la mémoire de l'ordinateur). Dans un programme, le symbole qui représente l'opération d'affectation est le signe égal (=). L'instruction d'affectation s'écrit en donnant l'identificateur de la variable, suivi du symbole d'affectation, puis de la valeur et enfin d'un point virgule, c'est-à-dire (avec une valeur littérale) :

identificateur = valeur littérale ;

Pour que cette instruction soit correcte, c'est-à-dire acceptée par le compilateur, **il faut que le type de la valeur littérale soit compatible avec celui de la variable**. Il est par exemple impossible de placer un réel dans une variable de type entier car la méthode de stockage n'est pas la même. De façon générale, le seul moyen de placer une valeur littérale dans une variable est que cette valeur soit d'un type "plus petit" que celui la variable. Nous verrons dans la section 2.2.4 le sens à

accorder à l'expression "plus petit". Notons que la valeur précédente contenue dans la variable est perdue car elle est remplacée par la nouvelle valeur.

REMARQUE

Entre les différents éléments d'une affectation, on peut mettre autant de fois le caractère d'espacement qu'on le souhaite. Ainsi `toto=2 ;` est-il aussi correct que `toto  =  2  ;`. Comme nous l'avons déjà dit à la section 1.3.4, le compilateur ne fait pas la différence entre un espace et une combinaison d'espaces, de tabulations et de passage à la ligne.

Il faut bien comprendre qu'une variable ne peut stocker qu'**une seul valeur**, comme l'illustre l'exemple suivant :

Exemple 2.5 :

Considérons le programme suivant :

```
int x;
x = 2;
x = 3;
```

Quelle est la valeur contenue dans `x` après l'exécution de la troisième ligne ? C'est bien entendu `3`. En effet, le processeur exécute les instructions dans l'ordre du programme. Il commence par créer une variable, puis place la valeur `2` dans cette variable. Ensuite, il place la valeur `3` dans la même valeur. L'ancienne valeur est tout simplement **remplacée** par la nouvelle et n'existe donc plus.

2.2.3 Affectation du contenu d'une variable à une autre variable

L'affectation ne se contente pas de permettre de placer une valeur dans une variable. Elle permet aussi de recopier le contenu d'une variable (la valeur qu'elle contient) dans une autre variable.

Exemple 2.6 :

Pour une raison quelconque, nous souhaitons échanger les valeurs contenues dans deux variables distinctes. Voici comment nous allons procéder :

```
int premier, second, temporaire;
premier = 2;
second = 1;
temporaire = premier;
premier = second;
second = temporaire;
```

Les trois premières lignes déclarent les variables. Les deux suivantes affectent à ces variables leur valeurs initiales. Quand nous saurons demander à l'utilisateur de taper au clavier une valeur numérique (cf le chapitre 3), l'exemple deviendra moins artificiel, mais pour l'instant, cette présentation est indispensable.

Les lignes suivantes constituent la partie nouvelle. La première d'entre elles indique au processeur de placer dans la variable `temporaire` la valeur que contient la variable `premier`. De ce fait, après l'exécution de l'instruction, `temporaire` contient la valeur 2. Ensuite, l'instruction suivante indique au processeur de placer dans la variable `premier` la valeur contenue dans la variable `second`. Après cette instruction, la variable `premier` contient la valeur 1. Ceci n'a aucune incidence sur la variable `temporaire` qui est bien sûr indépendante de la variable `premier`. Elle contient donc toujours la valeur 2. La dernière instruction permet alors de placer la valeur contenue dans `temporaire` dans la variable `second`. A la fin du programme, la variable

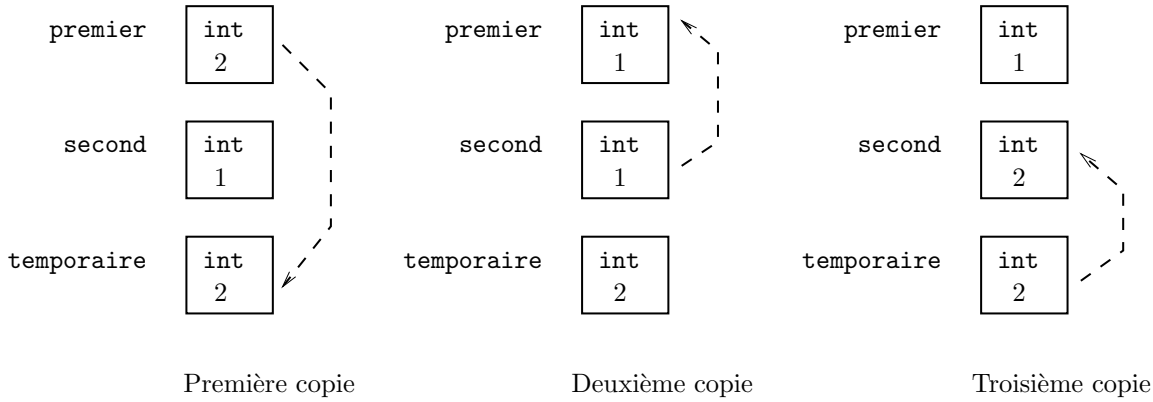

FIG. 2.3 – Illustration de l'échange de deux variables

`premier` contient donc 1 et la variable `second` contient 2. On a bien réussi à échanger les valeurs contenues dans les deux variables. Tout ce processus est illustré par la figure 2.3.

Nous apprenons grâce à cet exemple deux choses :

1. pour placer la valeur contenue dans une variable dans une autre variable, il suffit d'écrire le nom de la variable destinataire, suivi du signe égal, suivi du nom de la variable source, suivi d'un point virgule, ce qui donne la forme générale suivante :

   identificateur destinataire `=` *identificateur source* `;`

2. **les variables sont complètement indépendantes les unes des autres** : la modification de la valeur contenue dans une variable n'a aucune incidence sur les valeurs contenues dans les autres variables du programme.

REMARQUE

On utilisera souvent l'abus de langage qui consiste à dire *la valeur d'une variable* plutôt que *la valeur du contenu d'une variable*. En effet, le contenu d'une variable est un ensemble de *bit*s qui est interprété par le processeur afin de donner une valeur.

2.2.4 Compatibilité des types

Types numériques

Nous avons évité jusqu'à présent de considérer le problème suivant : que se passe-t-il si on tente de placer une valeur (ou le contenu d'une variable) de type `float` dans une variable de type `double` ? En fait, le type `double` est plus précis que le type `float`, donc il serait naturel d'autoriser une telle conversion. C'est fort heureusement le cas. Cette opération de conversion s'appelle une **promotion numérique**.Il est toujours possible de placer un `float` dans un `double`, et de même, il est toujours possible de placer un entier de "petite taille" dans un entier plus grand (par exemple un entier de type `int` dans une variable de type `long`).

Il est aussi possible de placer n'importe quel entier dans une variable de type réel (`float` ou `double`) bien que ceci puisse entraîner une perte d'information. En effet, les entiers de type `long` possèdent 19 chiffres significatifs, alors que les `float` en comptent environ 8 et les `double` 15 (pour plus de précision, voir la section 2.4.2).

De façon générale, il est impossible de faire des affectations dans l'autre sens, comme par exemple placer le contenu d'une variable de type `int` dans une variable de type `byte`. Les types numériques

sont donc rangés dans l'ordre suivant :

`byte<short<int<long<float<double`

Quand un type est "plus petit" qu'un autre dans cet ordre, on dit qu'il est **moins général** que l'autre. Par exemple, `int` est moins général que `float`.

On peut toujours placer une valeur d'un type donné dans une variable d'un type plus général (ou égal) dans cet ordre. Il est en général impossible de placer une valeur d'un type donné dans une variable d'un type strictement moins général.

Autres types

Le type `boolean` est compatible seulement avec lui-même. Cela signifie qu'il est strictement impossible de placer une valeur de type `boolean` dans une variable d'un type autre que `boolean`. De la même façon, seule une valeur de type `boolean` peut être placée dans une variable de type `boolean`.

La situation des variables de type `char` est plus complexe et sera traitée à la section 2.5.3. Pour simplifier on peut dire que seule une valeur de type `char` peut être placée dans une variable de type `char`. Par contre, on peut placer une valeur de type `char` dans une variable de type `int` (et donc dans une variable d'un type "plus grand" que `int`).

Exemple 2.7 :

Voici un exemple de programme incorrect car ne respectant pas les types :

Incorrect

```
1 public class Incorrect {
2   public static void main(String[] args) {
3     double x;
4     x=2;
5     int y;
6     y=x;
7   }
8 }
```

Contrairement à ce qu'on pourrait croire, la déclaration de la variable `y` après une instruction d'affectation ne pose aucun problème. On doit simplement avoir déclarée une variable avant de l'utiliser. Rien n'oblige à déclarer toutes les variables au début du programme. Le problème vient en fait de la ligne 6 car on tente de placer le contenu d'une variable de type `double` dans une variable de type `int`, ce qui n'est pas possible. Le programme n'est donc pas compilable et le compilateur indique le message d'erreur suivant[4] :

Erreur de compilation

```
Incorrect.java:6: possible loss of precision
found   : double
required: int
    y=x;
      ^
1 error
```

[4]Dans tout cet ouvrage, les messages d'erreur sont ceux obtenus avec les compilateurs fournis par `Sun` (cf http://www.java.sun.com/), plus précisément par la version 1.3.

Le message donné par le compilateur est assez précis, puisqu'il indique la raison profonde de l'erreur. En effet, *possible loss of precision* peut se traduire par "perte de précision possible". Le compilateur indique donc qu'il n'accepte pas l'affectation car, dans certains cas, la conversion d'un réel en un entier entraîne une perte d'information et donc une perte de précision dans la représentation de la valeur convertie.

Voici maintenant un exemple d'erreur légèrement différente :

Exemple 2.8 :

On considère le programme suivant :

IncorrectBoolean

```
1 public class IncorrectBoolean {
2   public static void main(String[] args) {
3     boolean b;
4     b=2;
5   }
6 }
```

Le compilateur refuse l'affecation d'une valeur entière dans une variable de type `boolean`. Il indique le message d'erreur suivant :

Erreur de compilation

```
IncorrectBoolean.java:4: incompatible types
found   : int
required: boolean
    b=2;
      ^
1 error
```

On constate que le message est différent de celui obtenu pour l'exemple précédent. Le compilateur indique ici simplement que les types intervenant dans l'affectation ne sont pas compatibles.

2.2.5 Déclaration avec initialisation

Il est possible d'utiliser la déclaration d'une variable pour donner directement la valeur qu'elle va prendre. Ce procédé est particulièrement utile. En effet, **une variable ne possède pas de valeur initiale** et il est **interdit** de l'utiliser tant qu'on ne lui a pas donné une valeur par une affectation.

Exemple 2.9 :

Le programme suivant est incorrect (car la valeur de `x` n'a pas été définie) :

PasDeValeur

```
1 public class PasDeValeur {
2   public static void main(String[] args) {
3     int x,y;
4     y=x;
5   }
6 }
```

Quand on tente de compiler ce programme, le compilateur produit le message d'erreur suivant :

Erreur de compilation

```
PasDeValeur.java:4: variable x might not have been initialized
    y=x;
      ^
1 error
```

Étudions maintenant un exemple de déclaration qui donne une valeur initiale à la variable déclarée :

Exemple 2.10 :

Reprenons l'exemple 2.6. Les trois premières lignes peuvent être réécrites de la façon suivante :

```
int premier=2 ;
int second=1 ;
int temporaire ;
```

et même de façon encore plus compacte :

```
int premier=2,second=1,temporaire ;
```

Le principe de ces déclarations est simple. Le signe égal et la valeur littérale qui suivent un identificateur indiquent que la variable ici déclarée prend comme valeur initiale la valeur littérale indiquée.

La forme générale des déclarations avec initialisation est la suivante :

type identificateur = valeur littérale ;

2.3 Calculs

L'une des tâches que l'ordinateur réalise le mieux est le calcul. En fait, les premiers ordinateurs n'étaient rien de plus que des machines à calculer très puissantes. Le but de notre apprentissage est entre autre de pouvoir réaliser des programmes de calcul numérique et il est donc important de bien comprendre les méthodes élémentaires permettant de faire calculer l'ordinateur abstrait. C'est le but de cette section.

2.3.1 Expressions arithmétiques

Le principal avantage de l'ordinateur face à certaines calculatrices est d'avoir une syntaxe simple pour les calculs simples. En d'autres termes, quand on veut faire exécuter un calcul à un ordinateur, il suffit en général de recopier la formule mathématique correspondante. On écrit alors une **expression**.

Exemple 2.11 :

Les lignes suivantes comportent chacune une expression :

```
2*(3+4)/1.5
2/(12+34-2.3)
(3<56) && (2>1)
5 > 123
```

Les deux premières lignes sont des expressions numériques classiques alors que les deux autres sont des expressions logiques (sur lesquelles nous reviendrons dans la section suivante).

En fait, toute expression arithmétique classique peut être traduite en une expression informatique. Il s'agit simplement de noter `*` le symbole de multiplication et `/` le symbole de division. Malheureusement, il est impossible d'avoir en `Java` une présentation aussi claire que celles qu'on obtient en mathématiques, comme en utilisant par exemple une barre de fraction. De ce fait, une formule `Java` est souvent moins lisible qu'une formule mathématique. Notons de plus que la suppression du symbole de multiplication, très pratique en mathématiques, est impossible dans une expression informatique. Si on veut calculer 2`x`, on est obligé d'écrire `2*x`.

Notons pour finir qu'on peut utiliser l'opérateur `%`. Celui-ci désigne l'opération qui donne pour résultat le reste de la division euclidienne du premier opérande par le deuxième. Par exemple la valeur de `5%3` est `2`. Remarquons que `5%3` se lit "cinq modulo trois".

REMARQUE

Il faut noter l'absence d'un opérateur de calcul de puissance. On ne peut donc pas écrire quelque chose de la forme `5^2` pour obtenir `5*5`.

2.3.2 Expressions logiques

Les expressions mathématiques classiques produisent un résultat numérique entier ou réel. L'ordinateur ne se contente pas d'effectuer ce genre de calcul. Il peut aussi faire des *calculs logiques*. Il s'agit alors de prendre une proposition logique comme par exemple `3 < 2` et de calculer sa *valeur de vérité*. La valeur de vérité d'une expression logique est *vrai* ou *faux*, c'est-à-dire `true` ou `false` en anglais. Une telle valeur est de type booléen (`boolean`).

Pour former des expressions logiques, on dispose *d'opérateurs de comparaison*. Les principaux opérateurs sont les suivants (ils permettent de comparer entre elles des valeurs littérales) :

Symbole	Signification
`<`	strictement inférieur
`<=`	inférieur ou égal
`==`	égal
`!=`	différent
`>=`	supérieur ou égal
`>`	strictement supérieur

La possibilité de *comparer* entre elles des valeurs numériques est un peu limitée. C'est pourquoi le processeur est capable de *combiner* entre elles des valeurs de vérité, par l'intermédiaire d'opérateurs logiques. Plus précisément, nous disposons des opérateurs suivants :

Symbole	Signification
`!`	*non* logique
`&&`	*et* logique
`||`	*ou* logique
`^`	*ou* exclusif

La valeur de vérité d'une expression utilisant un des opérateurs logiques est déterminée grâce à la table de vérité qui suit (dans laquelle "ou-ex" est une abréviation pour le ou exclusif) :

Opérande a	Opérande b	a et b	a ou b	a ou-ex b	non a
vrai	vrai	vrai	vrai	faux	faux
vrai	faux	faux	vrai	vrai	faux
faux	vrai	faux	vrai	vrai	vrai
faux	faux	faux	faux	faux	vrai

L'expression `(3 < 56) && (2 > 1)` de l'exemple 2.11 est donc une expression logique dont la valeur est `true`. En effet, `&&` représente un *et* logique. Or 3 est plus petit que 56 **et** 2 est plus grand que 1, donc les deux sous-expressions sont vraies **en même temps** et il en est de même de leur conjonction.

REMARQUE

Il faut faire attention de ne pas confondre l'opérateur `&` avec l'opérateur *et* logique `&&` (de même il ne faut pas confondre `|` avec `||`). Les deux opérateurs existent en `Java` et ne fonctionnent pas de la même façon, comme nous le verrons à la section 2.4.3. Pour éviter toute erreur, le plus simple est de n'utiliser que les opérateurs `&&` et `||`.

2.3.3 Affectation de la valeur d'une expression à une variable

De la même façon qu'on peut affecter une valeur littérale à une variable, on peut affecter la valeur d'une expression à une variable, comme le montre l'exemple suivant :

Exemple 2.12 :

Le programme suivant place dans différentes variables des expressions :

```
int toto ;
double valeur ;
boolean truth ;
truth = (2 == 3) || ( (3 > 4) || (!(2 == 5)) ) ;
valeur = 2/(12+34-2.3) ;
toto = 124+ 2*45 ;
```

Il est important de bien comprendre les éléments suivants :

- on place dans la variable la *valeur* de l'expression et pas l'expression elle-même. En effet, l'expression est un morceau de programme et ne constitue pas une valeur. Il faut donc que le processeur calcule la valeur de l'expression (c'est-à-dire qu'il **évalue** l'expression) avant de pouvoir placer celle-ci dans la variable considérée ;
- comme nous l'avons vu précédemment, le processeur ne peut pas affecter n'importe quelle valeur à une variable. Il faut que les types soient compatibles. On devine donc qu'une expression possède un type. Nous reviendrons à la section 2.3.5 sur ce problème délicat.

2.3.4 Expressions avec variables

Nous nous sommes pour l'instant intéressé aux expressions constantes, c'est-à-dire ne faisant pas intervenir des variables. Or, rien n'empêche d'utiliser des variables dans les expressions, comme le montre l'exemple suivant.

Exemple 2.13 :

Reprenons l'exemple 2.4 qui souhaitait calculer une moyenne. Nous obtenons le programme suivant :

```
double premier, second, moyenne ;
premier = 2.3434 ;
second = -23.4e-1 ;
moyenne = (premier + second) / 2 ;
```

Quel est le sens de la dernière ligne ? Nous remarquons tout d'abord une expression qui comporte des noms de variables. Comme à chaque variable correspond une valeur, il semble raisonnable de remplacer chaque identificateur par la valeur de la variable correspondante. Ainsi

l'expression devient-elle la suivante :

```
moyenne = ( 2.3434 + (-23.4e-1)) / 2 ;
```

L'interprétation est alors évidente, puis qu'on a retrouvé une expression constante.

De façon générale, une expression est une formule mathématique faisant apparaître des valeurs littérales, des opérateurs (dont les parenthèses) et des identificateurs.

L'interprétation à donner à une expression comportant des identificateurs de variables est très simple : on se contente de remplacer chaque identificateur par la valeur de la variable considérée. Ceci nous permet d'obtenir une expression simple dont on peut alors calculer la valeur. Le processus de calcul s'appelle l'**évaluation** de l'expression. On peut noter que le processus décrit à la section 2.2.3 pour l'affectation du contenu d'une variable à une autre est un cas particulier de l'évaluation d'une expression. En effet, si `x` est une variable, écrire `x` revient à donner une expression dans laquelle seule la variable `x` apparaît. La valeur de cette expression est par définition le contenu de la variable.

2.3.5 Type d'une expression

Comme nous l'avons évoqué précédemment, les expressions possèdent un type. Pour vérifier que le résultat d'une expression est bien compatible avec le type de la variable dans laquelle on souhaite placer sa valeur, l'ordinateur doit déterminer le type d'une expression. Pour ce faire, il applique les règles suivantes :

1. si l'expression est une valeur littérale ou une variable, elle a pour type celui de la valeur ou de la variable (pour le type des valeurs littérales, on se reportera à la section 2.2.1) ;
2. si l'expression est composée, l'ordinateur détermine l'opérateur le *moins* prioritaire. Il détermine d'abord le type des arguments de cet opérateur. Il applique alors la table suivante pour obtenir le type de l'expression :

Opérateur	type du résultat
`+`, `-`, `*`, `/` et `%`	type le plus général
comparaison (`<`, `>`, etc.)	`boolean`
`&&`, `||`, `!` et `^`	`boolean`

   Pour déterminer le type le plus général, l'ordinateur applique les règles suivantes :

   (a) si *au moins* une opérande est de type `double`, le type le plus général est `double` ;

   (b) sinon, si *au moins* une opérande est de type `float`, le type le plus général est `float` ;

   (c) sinon, si *au moins* une opérande est de type `long`, le type le plus général est `long` ;

   (d) sinon, le type le plus général est `int`.

   En fait, la règle est très simple : le type le plus "grand" (au sens de l'ordre indiqué à la section 2.2.4) l'emporte. La seule subtilité vient à ce niveau du fait que le plus petit type entier considéré est le type `int`.

Il est très important de noter une chose : la détermination du type d'une expression se fait **à la compilation** du programme. On dit d'ailleurs que le programme est **typé statiquement**[5]. Cela signifie que le type est déterminé sans faire les calculs (qui seront effectués au moment de l'**exécution** du programme). Donnons un exemple.

[5] En référence à l'opposition statique/dynamique présentée à la section 1.2.3.

Exemple 2.14 :

Considérons le programme suivant :

TypageStatique

```
1 public class TypageStatique {
2   public static void main(String[] args) {
3     int x;
4     float u,v;
5     u=4;
6     v=2;
7     x=u/v;
8   }
9 }
```

Ce programme n'est pas correct. En effet, d'après les règles de détermination du type d'une expression, on doit chercher le type le plus général entre ceux de `u` et de `v`. Or, ces deux variables sont de type `float` donc le résultat (l'expression) est de type `float` et ceci quelle que soit sa valeur effective. Ici par exemple, le résultat est entier et pourrait même être placé dans un `byte`. Il n'en reste pas moins que l'affectation de la dernière ligne est incorrecte. Le compilateur produit d'ailleurs le message suivant :

ERREUR DE COMPILATION

```
TypageStatique.java:7: possible loss of precision
found   : float
required: int
    x=u/v;
       ^
1 error
```

Un autre point est **extrêmement important**. Quand on divise entre eux deux entiers, les règles précédentes nous indiquent que le résultat est entier, ce qui signifie que l'opération renvoie le quotient (au sens de la division euclidienne) du premier opérande et du deuxième. Le résultat de `5/2` n'est donc pas `2.5` qui est de type `double` mais bien `2` (on pourra se reporter à l'exemple 2.15 pour quelques exemples).

REMARQUE

On remarque cependant que les règles énoncées ci-dessus ont des conséquences troublantes :

– Considérons par exemple le programme suivant :

TypageShort

```
1 public class TypageShort {
2   public static void main(String[] args) {
3     short a=1,b=1,c;
4     c=a+b;
5   }
6 }
```

Ce programme n'est pas correct. En effet, d'après les règles de calcul du type, l'expression `a+b` donne un résultat de type `int` qui ne peut donc pas être placé dans une variable de type `short`. Ceci signifie que les types `short` et `byte` ne sont pas adaptés pour le calcul et doivent plutôt être réservés au stockage d'information[6].

[6] Comme nous l'avons déjà dit, le type `int` s'impose dans la pratique comme le type entier le plus utile.

– Le problème est que dans l'exemple précédent, on a accepté comme correcte l'affectation `a=1`. D'ailleurs, le compilateur donne le message d'erreur suivant :

ERREUR DE COMPILATION

```
TypageShort.java:4: possible loss of precision
found   : int
required: short
    c=a+b;
      ^
1 error
```

Il est donc clair que le compilateur accepte la ligne `3`. Or, la valeur littérale `1` est de type `int` d'après la section 2.2.1 qui ne peut donc pas être placée dans une variable de type `short`. Comme nous l'indiquions justement dans cette section, il est cependant possible de placer quand même cette valeur dans un `short` car elle entre dans l'intervalle possible pour ce type. En fait, `Java` fait une différence entre les **expressions constantes** et les autres. On rappelle qu'une expression constante est une expression qui ne fait pas apparaître de variables mais seulement des valeurs littérales et des calculs. Quand un programme `Java` comporte des expressions constantes de type entier, `Java` autorise toutes les affectations compatible non pas simplement avec les types, mais aussi avec les intervalles de définition de ces types entiers. Pour revenir à l'exemple précédent, on peut donc écrire `c=1+1 ;`, ou même `c=100000-99999 ;`, car le compilateur va remplacer les expressions constantes par leur valeur qui respectent les contraintes du type `byte`.

2.3.6 Priorité des opérateurs

L'ordre des calculs en `Java` est proche de l'ordre utilisé conventionnellement en mathématiques. Voici un tableau des opérateurs classés dans l'ordre **décroissant** de priorité :

Niveau	Opérateur(s)
1	`++`, `--`
2	`-` (unaire), `+` (unaire), `!`, `(`*type*`)`
3	`*`, `/`, `%`
4	`+`, `-`
5	`<`, `>`, `<=`, `>=`
6	`==`, `!=`
7	`^`
8	`&&`
9	`||`
10	`=`, `+=`, `-=`, `*=`, `/=`

Certains opérateurs apparaissant dans ce tableau seront étudiés dans les sections suivantes :
– les opérateurs `++` et `--` sont étudiés à la section 2.5.2 ;
– la famille d'opérateurs `(`*type*`)` est étudiée à la section 2.5.1 ;
– les opérateurs `+=`, `-=`, etc. sont étudiés à la section 2.5.2 ;

REMARQUE

Les règles de priorité permettent d'interpréter des calculs très complexes ne faisant pas intervenir de parenthèses. Il est vivement **déconseillé** d'utiliser ce genre d'écriture. On peut par exemple

utiliser l'opérateur moins unaire pour écrire `2*-1` qui sera interprété comme `2*(-1)`. La deuxième écriture est nettement plus claire.

Exercice corrigé

En tenant compte des règles de typage et des priorités, indiquez dans le programme suivant quelles sont les affectations correctes et incorrectes, en justifiant votre réponse.

```
1  double x=1,y=2 ;
2  boolean b=x<y ;
3  float z=2.5f ;
4  int u,v ;
5  u=y/x ;
6  u=3 ;
7  v=5L ;
8  v=3*-u ;
9  b=z<x ;
```

Pour résoudre cet exercice, il faut procéder de façon très rigoureuse en appliquant les règles énoncées dans ce chapitre. La première ligne est clairement correcte. En effet, les deux valeurs littérales sont de type `int` et peuvent donc être placées dans des variables de type `double`. La deuxième ligne est tout aussi correcte. En effet, elle réalise une comparaison entre deux `double`s, ce qui donne un `boolean`, qui peut donc être placé dans une variable du même type. La troisième ligne est correcte car elle place une valeur littérale de type `float` (à cause du `f`) dans une variable du même type. La quatrième ligne est bien entendu correcte. La ligne 5 est incorrecte. En effet, l'expression faisant intervenir deux `double`s, sa valeur est de type `double` (même si c'est ici un entier) et on ne peut pas placer un `double` dans une variable de type `int`. La ligne 6 est bien sûr correcte. Par contre, ce n'est pas le cas de la 7. En effet, la valeur littérale est de type `long` (à cause du `L`) et on ne peut pas placer une telle valeur dans une variable de type `int`. La ligne suivante est correcte, malgré sa syntaxe un peu déroutante. Comme l'opérateur unaire `-` est plus prioritaire que `*`, on peut sans rien changer remplacer `3*-u` par `3*(-u)`. Le type de cette expression est `int` car elle ne fait apparaître que des `int`s et de ce fait, l'affectation est correcte. Enfin, la ligne 9 est aussi correcte car elle place le résultat d'une comparaison (de type `boolean`) dans une variable adaptée. Notons qu'il est ainsi parfaitement possible de comparer des variables de types numériques différents (ici `z` est un `float` alors que `x` est un `double`).

Il faut être particulièrement attentif aux interactions entre typage et priorité, comme le montre l'exemple suivant :

Exemple 2.15 :

On considère le morceau de programme suivant :

```
double u=3 ;
int v=3 ;
```

On forme des expressions en utilisant les variables `u` et `v`. Le tableau suivant indique pour

chaque expression formée le type et la valeur du résultat, ainsi qu'une version de l'expression plus claire grâce à l'utilisation de parenthèses :

Expression	type	valeur	simplifiée
`u/v`	`double`	`1.0`	`u/v`
`1/v*u`	`double`	`0.0`	`(1/v)*u`
`1/u*v`	`double`	`1.0`	`(1/u)*v`
`u*1/v`	`double`	`1.0`	`(u*1)/v`
`v/3/u`	`double`	`0.33333...`	`(v/3)/u`
`2/3*v`	`int`	`0`	`(2/3)*v`
`2/3*u`	`double`	`0.0`	`(2/3)*v`
`2.0/3*v`	`double`	`2.0`	`(2.0/3)*v`

On remarque que certaines expressions très proches, comme par exemple `1/v*u` et `1/u*v` donnent des résultats différents. Pour cet exemple, l'interprétation est simple. Dans le premier calcul, on effectue d'abord `1/v`, une opération en entier qui donne pour résultat `0` (le quotient de la division de 1 par 3). La multiplication par 3 donne toujours 0. Dans la deuxième version, on calcule d'abord `1/u`, ce qui donne un calcul en `double`. On obtient donc environ $\frac{1}{3}$ qui, multiplié par 3, donne bien 1. C'est toujours le problème du calcul en `int` qui explique la différence de résultat entre `2.0/3*u` et `2/3*u`. Cette dernière expression est très intéressante car elle montre que la conversion en `double` est bien effectuée lors du *second* calcul (la multiplication par `u`), alors que le premier calcul se fait en `int`.

2.4 Mécanismes évolués de l'évaluation

2.4.1 Les calculs impossibles

Calculs avec des entiers

Pour certaines expressions, il n'est pas possible de définir une valeur pertinente. Il est par exemple impossible d'effectuer une division par zéro. On doit tout d'abord noter que le **compilateur** accepte tous les calculs qui sont syntaxiquement corrects et qui respectent les types. De ce fait, le programme suivant est accepté :

Division

```
1 public class Division {
2   public static void main(String[] args) {
3     int i=1/0;
4   }
5 }
```

Par contre, le **processeur** refuse d'effectuer le calcul car il ne peut bien entendu pas définir de résultat. Au moment de l'exécution du programme, on obtient le message d'erreur suivant :

Erreur d'exécution

```
java.lang.ArithmeticException: / by zero
        at Division.main(Division.java:3)
```

Si on tente d'effectuer indirectement une division par zéro (essentiellement en calculant le reste de la division d'un entier par zéro), on obtient le même message.

REMARQUE

Il faut bien se garder de confondre une erreur de compilation et une erreur d'exécution. La division par zéro provoque une erreur à l'exécution du programme, ce qui signifie que c'est le processeur qui s'aperçoit de l'erreur, par opposition aux erreurs statiques, découvertes par le compilateur.

Calculs avec des réels

La seule difficulté concernant les calculs impossibles est que `Java` ne considère pas les entiers et les réels de la même façon. Comme nous venons de le voir, une division par zéro utilisant des entiers provoque une erreur à l'exécution du programme. Par contre, ce n'est pas le cas pour les réels (types `double` et `float`). En effet, le codage des réels prévoit des valeurs spéciales correspondant à $+\infty$ et à $-\infty$. De plus, une autre valeur particulière indique que le "réel" considéré ne correspond à aucun réel (par exemple quand on divise zéro par lui-même). De ce fait, **un calcul réel est toujours possible** et ne provoque donc jamais d'erreur à l'exécution du programme.

2.4.2 Les calculs incorrects

Calculs avec des entiers

Comme nous l'avons indiqué à la section 2.1.2, chaque type entier est associé à un intervalle borné de $\mathbb{Z}$. Il est donc possible lors d'un calcul d'obtenir un résultat en dehors de l'intervalle, alors que les opérandes du calcul étaient contenues dans l'intervalle. Considérons le programme suivant :

```
int i=2147483647;
i=i+1;
```

Comme la valeur initiale de `i` est la plus grande valeur pour un `int`, on peut se demander si le processeur accepte d'effectuer l'addition de la deuxième ligne. C'est le cas, mais le résultat est inattendu. En effet, la valeur de `i` après l'exécution de la deuxième instruction est `-2147483648`, soit la plus petite valeur possible pour un `int`. Notons avant toute chose que ceci n'est pas une conséquence du stockage dans `i`. C'est le calcul `i+1` qui donne ce résultat. En effet, si on écrit le programme :

```
int i=2147483647;
double x=i+1;
```

Le contenu de `x` après l'exécution de la deuxième ligne est bien `-2147483648.0` (en fait `-2.147483648E9`), soit la conversion en `double` du plus petit `int` acceptable.

L'explication du comportement de l'addition est relativement complexe car l'interprétation des calculs sur les entiers est étroitement liée au codage utilisé pour ceux-ci. Le codage des entiers positifs ou nul ne pose aucun problème : il s'agit de l'écriture en base 2 de l'entier considéré. Par exemple, 5 est représenté par `101`, nombre complété par 29 zéros dans le cas d'un `int` et `61` zéros dans le cas d'un `long`. Pour les entiers strictement négatifs, on utilise un codage plus complexe, qui possède l'avantage d'être facile à mettre en oeuvre par les processeurs. Pour coder un entier x négatif, on utilise la représentation en base 2 de l'entier $2^{32} + x$ pour les `int`s et celle de $2^{64} + x$ pour les `long`s. Ceci explique le codage de -30 qui est donc la représentation binaire de $2^{32} - 30$, soit de 4 294 967 266, c'est-à-dire :

`11111111111111111111111111100010`

On remarque que tous les codes de 0 à $2^{31} - 1$ sont utilisés pour les entiers positifs. Comme les entiers négatifs vont de -2^{31} à -1, on utilise pour eux les codes des entiers de 2^{31} (qui correspond

à $2^{32} - 2^{31}$) à $2^{32} - 1$. De la même façon, les entiers négatifs utilisent les écritures binaires des entiers compris au sens large entre 2^{63} et $2^{64} - 1$, quand on utilise des `long`s.

Pour comprendre les calculs sur les entiers, il faut d'abord noter que toute opération est définie modulo 2^{32} pour les `int`s et modulo 2^{64} pour les `long`s. Supposons que la variable `x` contienne la valeur u et que la variable `y` contienne la valeur v. `x op y` ne correspond pas à u `op` v, mais à

$$u \texttt{ op } v \left[2^{32}\right]$$

où `op` désigne un opérateur quelconque fonctionnant sur les entiers (multiplication, addition, etc.) et où a $[b]$ désigne le reste de la division de a par b. Pour les `long`s, c'est la même chose mais avec 2^{64} comme diviseur.

Considérons l'exemple de l'addition de deux entiers stockés dans des `int`s. Si la somme des deux entiers est inférieure à $2^{31} - 1$, il n'y a pas de problème, le modulo n'ayant pas d'effet. Si par contre le résultat est compris entre 2^{31} et $2^{32} - 1$, le modulo n'a toujours pas d'effet, mais le résultat n'est pas interprété correctement. En effet, pour le processeur, une représentation binaire correspondant à un entier compris entre 2^{31} et $2^{32} - 1$ est utilisée pour un entier négatif. De ce fait, la valeur considérée pour le résultat du calcul est obtenue en retranchant 2^{32} au vrai résultat. Ceci explique l'exemple présenté en introduction. En effet, 2147483647 correspond à $2^{31} - 1$. Quand on lui ajoute 1, on obtient donc 2^{31}. Pour le processeur, il s'agit du code d'un entier négatif, dont la valeur est obtenue en retranchant 2^{32}, ce qui donne exactement -2^{31}, soit -2147483648. Le phénomène est exactement le même avec les `long`s, en utilisant bien sûr les valeurs 2^{63} et 2^{64}.

Quand le dépassement est plus important, le modulo entre en jeu. Considérons par exemple le calcul suivant :

```
int i=2147483647;
i=i*10;
```

Après ce calcul, le contenu de `i` est -10. En effet, le reste de la division de 21 474 836 470 par 2^{32} est 4 294 967 286. Ce nombre est supérieur à 2^{31} et donc interprété comme un nombre négatif. Quand on lui soustrait 2^{32}, on obtient bien -10.

Du point de vue pratique, le résultat d'un calcul avec des entiers correspond en général à l'interprétation simple des opérateurs. Autrement dit, en général l'addition informatique correspond à l'addition mathématique. Les problèmes surviennent lors des dépassements, comme le premier exemple l'a montré. Pour comprendre intuitivement ce qui se passe, on peut considérer que les entiers informatiques possèdent une structure circulaire. Le nombre qui suit 2147483647 (c'est-à-dire $2^{31} - 1$) n'est pas 2147483648, mais -2147483648, c'est-à-dire -2^{31}. Si on ajoute 10 à 2147483647, le résultat n'est donc pas 2147483657, mais celui de $-2147483648 + 9$, car on "avance" de dix unités à partir de $2^{31} - 1$: la première unité nous fait atteindre -2^{31}, puis chaque unité est interprétée normalement. On obtient donc -2147483639. De la même façon, le nombre qui précède -2^{31} n'est pas $-2^{31} - 1$ mais $2^{31} - 1$. Ce point de vue est bien entendu applicable aux `long`s, à condition de remplacer 2^{31} par 2^{63}.

REMARQUE

Le mécanisme d'évaluation des calculs entiers peut sembler très étrange. En fait, il est parfaitement justifié par la représentation binaire des entiers. C'est l'efficacité qui prévaut dans cette représentation, au détriment de l'exactitude mathématique. Cela peut parfois engendrer des problèmes, mais en général, on peut raisonnablement supposer que les calculs sont exacts.

Calculs avec des réels

Nous avons indiqué dans la section 2.1.2 que les `float`s possèdent 7 à 8 chiffres significatifs, alors que les `double`s en offrent le double[7]. Il est temps de préciser cette notion.

Le nombre de chiffres significatifs correspond au nombre de chiffres nécessaires à l'écriture minimale du nombre considéré en notation scientifique. Si on considère par exemple le nombre 0.00025, on ne doit pas parler de 6 chiffres significatifs, mais de 2, car on peut l'écrire sous la forme $2.5\ 10^{-4}$. Considérons le programme suivant :

```
float x=0.00000000000000025f;
```

Avec une mauvaise compréhension des chiffres significatifs, on pourrait croire que cette affectation va donner un résultat faux. Mais en fait, ce nombre est simplement $2.5\ 10^{-16}$ et ne possède donc que 2 chiffres significatifs, ce qui ne pose aucun problème pour un `float`. Les réels informatiques sont représentés en **virgule flottante** (d'où le nom *float*) : cela signifie que la virgule (le point décimal en notation informatique) ne correspond pas toujours à la séparation entre la partie entière et la partie fractionnaire. L'utilisation des puissances de dix (la notation scientifique) correspond en fait à faire bouger la virgule, comme l'illustre l'exemple précédent. Cette représentation évite d'avoir à conserver dans la mémoire de l'ordinateur des zéros inutiles. Dans l'exemple proposé, on passe ainsi d'une écriture avec 18 chiffres à une écriture avec 4 chiffres (deux pour le nombre lui-même et 2 pour l'exposant).

Cette technique ne permet cependant pas de dépasser le nombre maximum de chiffres significatifs permis par le type utilisé, comme l'illustre l'exemple suivant :

```
float x=1.23456789123456789f;
```

Notons tout d'abord que le compilateur accepte sans problème l'affectation, même si le nombre proposé compte 18 chiffres significatifs. Par contre, après l'affectation, le contenu de `x` est 1.2345679. Ce nombre possède bien 8 chiffres significatifs et est une version approchée du nombre de départ. La règle de base à retenir avec les `float`s et les `double`s est que les calculs sont toujours **approximatifs**. Considérons par exemple le programme suivant :

```
double u=1;
double v=1e-20;
double w=u+v;
```

Le résultat mathématiquement correct est 1.00000000000000000001, soit un nombre à 21 chiffres significatifs. Or, les `double`s comptent 15 à 16 chiffres significatifs. Donc `w` contient une valeur arrondie du résultat correct, en l'occurrence 1.

Ce phénomène d'arrondi n'est pas le seul problème qu'on puisse rencontrer avec les réels. Il existe en effet un plus petit réel (positif) représentable, ainsi qu'un plus grand réel (positif) représentable. Voici ces valeurs :

type	valeur absolue minimale	valeur absolue maximale
`float`	$1.4\ 10^{-45}$	$3.4028235\ 10^{38}$
`double`	$4.9\ 10^{-324}$	$1.7976931348623157\ 10^{308}$

Quand un calcul donne un résultat trop grand, la valeur obtenue est un code spécial qui correspond à l'infini positif ou négatif (selon le signe de la valeur qui devait être obtenue). De même quand un calcul donne un résultat trop petit, la valeur obtenue un zéro positif ou négatif (selon le signe de la valeur qui devait être obtenue). Des règles précises régissent le comportement de ces valeurs spéciales. Par exemple, si on divise une valeur strictement positive par un zéro négatif, on obtient un infini négatif, etc. Comme nous l'avons dit dans la section précédente, quand le résultat du calcul ne peut pas être défini, le processeur utilise un code particulier.

[7] D'où leur nom, d'ailleurs.

REMARQUE

On peut se demander pourquoi les valeurs maximales et minimales des réels ne sont pas des valeurs "rondes". De même on peut se demander pourquoi le nombre de chiffres significatifs peut varier (7 ou 8 pour les `floats`, par exemple).

Comme pour le mécanisme complexe de calcul sur les entiers, tout ceci est lié au codage des réels. Ce codage est binaire et se représente donc naturellement par des puissances de 2. Par exemple, la valeur absolue d'un `float` s'écrit sous la forme $m\ 2^e$, où m est un entier positif strictement inférieur à 2^{24} et où e est un entier compris au sens large entre -149 et 104. Pour les `doubles`, m est strictement inférieur 2^{53}, alors que e est compris entre -1075 et 970.

De tout ce mécanisme complexe, il faut surtout retenir que les calculs donnent des résultats approximatifs, dans la limite de la précision indiquée pour chacun des deux types réels. Il faut d'ailleurs noter que la promotion numérique des entiers vers les réels peut poser problème, comme le montre le programme suivant :

```
int i=2147483647;
float y=i;
long j=9223372036854775807;
double x=j;
```

Après l'exécution du programme, le contenu de `y` est `2.14748365E9`, soit une approximation du contenu de `i`. De même, le contenu de `x` est `9.223372036854776E18`, qui est aussi une approximation de `j`. Le programme est bien entendu parfaitement correct, mais on est ici confronté aux limitations des types réels.

2.4.3 L'évaluation court-circuitée

Terminons cette section sur le calcul par une subtilité des expressions logiques. Considérons le programme suivant :

CourtCircuit

```
1 public class CourtCircuit {
2   public static void main(String[] args) {
3     int a=2,b=0;
4     boolean test=(a<1)&&(a/b==3);
5   }
6 }
```

Comme il comporte une division par zéro, ce programme devrait provoquer une erreur d'exécution. Or, ce n'est pas le cas. Le programme fonctionne parfaitement et la variable `test` contient, après l'exécution de l'affectation, la valeur `false`. Ceci est du à une technique appelée **évaluation court-circuitée**.

Le calcul de la valeur à placer dans `test` fait en effet intervenir l'opérateur `&&` qui correspond au *et* logique. Or, d'après la table de vérité (cf la section 2.3.2), le résultat d'un *et* logique est faux si l'une au moins des deux sous-expressions logiques est elle-même fausse. Or, quand le processeur évalue une expression, il procède de gauche à droite (en tenant bien sûr compte des priorité). Dans le programme étudié, le processeur commence donc par calculer `(a<1)`. Il obtient ainsi la valeur `false`. A ce moment, quelle que soit la valeur de l'expression `(a/b==3)`, le résultat du *et* logique est nécessairement `false`. Le principe de l'évaluation court-circuitée est **de ne pas évaluer** la deuxième expression si on peut s'en passer. Dans notre exemple, la valeur de `(a/b==3)` ne sera donc jamais calculée. Donc, il n'y aura pas de division par zéro et pas d'erreur d'exécution du programme.

Le processeur évalue aussi les *ou* logiques de façon court-circuitée. Il n'y a donc pas d'erreur d'exécution quand le processeur évalue L'expression `(2>1)||(3/0==2)`. En effet, la valeur de `(2>1)` est `true`, ce qui suffit à assurer que l'expression globale vaut `true`.

Il faut faire très attention de ne pas confondre les opérateurs `&` et `&&`. En effet, `&` correspond au *et* logique, mais demande une évaluation **non** court-circuitée. De la même façon `|` correspond au *ou* logique sans évaluation court-circuitée. Si on réécrit le programme `CourtCircuit` en utilisant l'opérateur `&`, on obtient :

SansCourtCircuit

```
1 public class SansCourtCircuit {
2   public static void main(String[] args) {
3     int a=2,b=0;
4     boolean test=(a<1)&(a/b==3);
5   }
6 }
```

Ce programme est parfaitement correct, mais son exécution provoque l'erreur suivante :

ERREUR D'EXÉCUTION

```
java.lang.ArithmeticException: / by zero
        at SansCourtCircuit.main(SansCourtCircuit.java:4)
```

REMARQUE

Dans la pratique, il est très rare d'avoir besoin des opérateurs `&` et `|` car l'évaluation court-circuitée est à la fois plus rapide et plus pratique. De ce fait, il faut bien retenir son principe.

Exemple 2.16 :

Voici un exemple d'application pratique de l'évaluation court-circuitée. Supposons données deux variables `x` et `y`. On souhaite former une expression logique vraie si et seulement si le contenu de `x` est divisible par le contenu de `y`. Naïvement, on peut écrire l'expression `x%y==0`, car le reste de la division de `x` par `y` est nul si et seulement si `x` est divisible par `y`. Mais on oublie ici le cas où `y` contient 0. Dans ce cas, la division est impossible et le programme plante. Pour éviter ce problème, on peut utiliser l'expression `y!=0 && x%y==0`. Grâce à l'évaluation court-circuitée, cette expression ne pose jamais de problème. Quand `y` est nul, elle s'évalue en `false`, sans avoir à effectuer la division problèmatique.

2.5 Compléments

2.5.1 Les conversions numériques

Comme nous l'avons vu dans les sections précédentes, il est impossible de placer dans une variable d'un type donné une valeur d'un type qui n'est pas compatible avec celui-ci. Pour les valeurs numériques, ceci se simplifie en disant qu'on ne peut placer une valeur dans une variable que si le type de la variable est plus général que celui de la valeur.

Cette interdiction n'est pas toujours pratique. Fort heureusement, il existe un moyen de la contourner, qui porte le nom de **conversion numérique**[8]. Considérons l'exemple suivant :

[8] On parle aussi de *transtypage* numérique pour insister sur le fait que la valeur change de type.

```
double x=3.5 ;
int i=(int)x ;
```

Quel en est le sens ? En fait, la présence de `(int)` devant la variable `x` demande au processeur de convertir la valeur de `x` (qui est donc de type `double`) en un entier (plus précisément un `int`).

De façon plus générale, on peut écrire `(`*type*`)(`*valeur*`)` pour convertir une valeur d'un type numérique[9] quelconque en une valeur du type (numérique) précisé devant. La seconde paire de parenthèses peut être omise quand on convertit une valeur littérale ou une valeur contenue dans une variable, mais pas en général car l'opération de conversion est **prioritaire** devant les calculs.

Exemple 2.17 :

Si on souhaite par exemple calculer le résultat de la division *non euclidienne* de deux entiers entre eux, on peut utiliser la conversion. Considérons par exemple le programme suivant :

```
int u=5,v=2 ;
double x,y ;
x=u/v ;
y=(double)u/v ;
```

Après l'exécution de ces différentes instructions, la variable `x` contient la valeur `2` alors que la variable `y` contient la valeur `2.5`. En effet, pour le premier calcul, `u` et `v` sont des entiers et leur division est euclidienne, même si le résultat est placé dans un réel. Par contre, dans la deuxième formule, on convertit `u` en un réel (un `double` pour être précis), puis on fait le calcul, qui par la règle du type le plus général fait donc intervenir des `double`s, ce qui permet d'obtenir le résultat de la division exacte plutôt que celui de la division euclidienne.

La conversion d'un entier en un réel ne pose pas de problème *a priori*. Par contre la conversion en sens inverse est plus problématique. Elle suit deux règles simples :

- **règle de dépassement** : la conversion d'un réel en un entier (`int`) est *saturante*. Si la valeur du réel n'entre pas dans l'intervalle représenté par un `int` (cf. section 2.1.2, page 16), la valeur entière résultat de la conversion est le maximum de cet intervalle si le réel est positif, ou le minimum s'il est négatif.
- **règle de troncature** : la valeur entière d'un réel est la partie qui précède la virgule. Il ne s'agit donc pas de la partie entière au sens mathématique[10]. Le résultat de `(int)3.5` est donc `3`, alors que celui de `(int)-3.5` est `-3`.

2.5.2 Opérateurs compacts

Il existe en `Java` des opérateurs spéciaux qui permettent de rendre un programme plus compact, plus efficace et souvent plus lisible. Ces opérateurs réalisent une combinaison entre une opération classique (comme une addition) et une affectation.

Dans le tableau qui les présente, `i` et `j` désignent des variables de type réel ou entier, telles que la valeur de `j` puisse être placée dans `i` :

Opération	Synonyme de
`i++ ;`	`i = i+1 ;`
`i- ;`	`i = i-1 ;`
`i += j ;`	`i = i+j ;`
`i -= j ;`	`i = i-j ;`
`i *= j ;`	`i = i*j ;`
`i /= j ;`	`i = i/j ;`

[9] J'insiste, ceci ne fonctionne pas avec le type `boolean` par exemple.

[10] On rappelle que la partie entière d'un réel x, notée $[x]$, est le plus grand entier inférieur ou égal à x, ce qui signifie que $[3.5] = 3$, alors que $[-3.5] = -4$.

On peut bien sûr remplacer `j` par une valeur ou une expression, à partir du moment où celle-ci est compatible avec `i`. La subtilité principale est la suivante : si on utilise par exemple `i *= 5+2 ;`, le processeur effectue l'opération `i = i*(5+2) ;` et non pas `i = i*5+2 ;`. L'expression qui apparaît à droite est évaluée avant que le calcul portant sur la variable (qui apparaît à gauche) soit effectué.

La deuxième subtilité concerne le typage. Supposons que `b` soit une variable de type `byte`. On sait que `b=b+2 ;` n'est pas une affectation valide car le type de `2` est `int` et donc `b+2` est aussi de type `int`. Cependant, `b+=2 ;` est correct. En effet, la véritable traduction de `b+=2 ;` n'est pas `b=b+2 ;` mais `b=(byte)(b+2) ;`. L'opération de conversion numérique est ajoutée automatiquement par `Java`.

REMARQUE

L'écriture `i=i++ ;` donne un résultat inattendu : l'effet de cette instruction est nul et la valeur de `i` reste inchangée. En fait, `i++` est une **expression** dont la valeur est celle contenue dans la variable `i` **avant son incrémentation**. De ce fait, écrire `i=i++ ;` provoque d'abord l'évaluation de l'expression, puis l'écriture du résultat dans la variable `i`. Or, l'évaluation de l'expression donne pour résultat l'ancienne valeur de `i` et a pour **effet de bord** d'ajouter 1 au contenu de `i`. Ensuite on place la valeur obtenue (l'ancienne valeur de `i`) dans `i`. Nous sommes donc revenus à notre point de départ !

Il est possible d'utiliser l'expression `++i` (de même que `-i`). L'effet sur la variable `i` est le même que celui de `i++`, on ajoute un à la valeur contenue dans `i`. Comme `i++`, `++i` est une expression, mais sa valeur est celle contenue dans la variable `i` **après** son incrémentation.

Nous verrons dans la suite du cours de nombreuses applications de ces opérateurs compacts.

2.5.3 Le type `char`

Contrairement à ce que le type `char` pourrait laisser croire, l'ordinateur abstrait n'est pas capable de manipuler *directement* des caractères : il doit traduire chaque caractère en une valeur numérique (entière) pour pouvoir le stocker dans sa mémoire. La convention de codage adoptée est celle définie par le consortium Unicode[11]. Elle consiste à associer à chaque caractère une valeur entière comprise entre 0 et 65535 (ce qui correspond à deux octets), ce qui permet de représenter l'alphabet romain, mais aussi de nombreux autres alphabets (arabe, cyrillique, grec, hébreux, thai, etc.).

Le code associé à chaque caractère n'est pas vraiment important. Par contre, les conséquences de l'utilisation de ce code sont importantes : le type `char` est un type entier comme les autres (c'est-à-dire au même titre que `int` et `short` par exemple). De ce fait, on peut donc placer une valeur de type `char` dans une variable de type entier suffisamment *grande*, c'est-à-dire de type `int` ou `long`[12]. Cependant, il est impossible de placer une valeur entière classique dans un `char` car les autres types entiers peuvent prendre des valeurs négatives. Par contre, on peut faire des opérations numériques sur les caractères : on peut ajouter deux `char`, le résultat étant la somme des codes Unicode des deux caractères de départ. Comme dans toutes les opérations entières, le résultat est de type `int`, ce qui oblige à le convertir en `char` pour obtenir un nouveau caractère. On peut par exemple effectuer la manipulation suivante :

```
char a='a';
a=(char)(a+1);
```

[11] http://www.unicode.org

[12] On remarque qu'il n'est pas possible de placer un `char` dans une variable de type `short` car ce dernier est limité à 32767. Ceci montre bien qu'utiliser le même nombre de cases en mémoire qu'un type est loin d'être suffisant pour être compatible avec lui. Par contre, il est bien entendu possible de placer un `char` dans une variable de type `float` ou `double`.

Le contenu de `a` est alors le caractère `b`. On remarque au passage que le code Unicode conserve en général l'ordre alphabétique, c'est-à-dire que le code d'une lettre est d'autant plus grand qu'elle est proche de la fin de l'alphabet. De plus, les codes se suivent comme dans l'alphabet.

Pour obtenir une valeur littérale de type `char` correspondant à un code Unicode particulier, on écrit par exemple `'\u0041'`. Les quatre chiffres qui suivent le `u` forment un nombre correspondant au code Unicode. Le nombre en question est codé en **hexadécimal** (c'est-à-dire en base 16).

REMARQUE

Dans la pratique, il est assez peu courant d'utiliser le codage des caractères, mais il est important de connaître son existence.

2.6 Conseils d'apprentissage

Ce chapitre est assez long et technique. Les exemples proposés restent assez artificiels car nous ne disposons pas des constructions nécessaires à l'écriture de véritables programmes. Le lecteur peut donc légitimement s'interroger sur l'importance relative des différents points abordés dans ce chapitre. Voici quelques éléments de réponse :

- Les deux instructions étudiées dans ce chapitre, la **déclaration de variable** et l'**affectation** sont les briques de base de tout programme (dans tout langage, d'ailleurs).
- La notion de **variable** est fondamentale dans tous les langages de programmation. Sans variable, on ne peut pas écrire de programme.
- La notion de **type** est tout aussi fondamentale. Pour écrire un programme, il faut impérativement comprendre le **typage**, en particulier celui des expressions, en gardant à l'esprit qu'il est réalisé **statiquement**.
- Dans la pratique, la maîtrise parfaite des types `int`, `double`, `boolean` et `char` est indispensable. Il faut en particulier être très attentif aux **divisions** qui n'ont pas le même résultat selon qu'on travaille en entier ou en réel. Cette subtilité est source d'erreurs fréquentes.
- Le concept d'**évaluation d'une expression** et son interaction avec le typage doivent être bien compris.
- L'**évaluation court-circuitée** est très utile dans les expressions logiques et doit être maîtrisée.
- Pour éviter des problèmes faisant intervenir les priorités des opérateurs, il est vivement conseillé d'utiliser des **parenthèses** dans les expressions complexes.
- Le respect des **conventions** pour les noms de variables est crucial pour obtenir des programmes lisibles.

CHAPITRE 3

Utilisation des méthodes et des constantes de classe

Sommaire

Introduction

Dans les chapitres précédents, nous avons eu une approche assez théorique de la programmation pour une raison simple : nous n'avons pour l'instant aucun moyen de communiquer avec un programme. Nous ne savons pas comment faire en sorte que le processeur demande une valeur à l'utilisateur d'un programme. Nous ne savons pas comment afficher le résultat d'un calcul. Le but de ce chapitre est de combler ces lacunes afin de pouvoir écrire des programmes un peu plus réalistes.

L'interaction d'un programme avec l'utilisateur est une des tâches les plus complexes à réaliser. Pour ne pas être confrontés à cette difficulté, nous allons utiliser des programmes déjà existants. En `Java`, il est possible d'utiliser dans un programme un composant d'un autre programme. Le mécanisme employé est l'**appel de méthode**. Une méthode est un composant de programme, une suite d'instructions, qu'on peut utiliser dans un autre programme pour réaliser une tâche, sans avoir besoin de recopier les instructions qui constituent la méthode. L'appel de méthode obéit à des règles de **typage** et d'**évaluation** très similaires à celles utilisées pour les variables.

Nous verrons dans ce chapitre comment utiliser des méthodes et nous donnerons une première liste (non exhaustive, bien entendu) de méthodes utiles. Nous verrons ainsi qu'il est possible d'effectuer des calculs scientifiques classiques en `Java` (fonctions trigonométriques, etc.). Nous aborderons le point très important de l'interaction d'un programme avec l'utilisateur, en donnant des méthodes d'**affichage** et des méthodes de **saisie**. Nous présenterons la notion de **constante** qui simplifie l'écriture de certains programmes et pour finir, nous indiquerons comment utiliser les **paquets**.

3.1 Les méthodes de classe

3.1.1 Définition

Comme nous l'avons dit en introduction, une méthode est une suite d'instructions. Plus précisément, on peut donner la définition suivante :

Définition 3.1 *Une* ***méthode de classe*** *est une* suite d'instructions. *Elle est désignée par un* identificateur *et appartient à une* classe. *Elle utilise éventuellement des* paramètres *et produit éventuellement comme* résultat *une valeur.*

Il nous faut bien entendu définir la notion de classe :

Définition 3.2 *Une* ***classe*** *est un groupe d'élements* `Java`. *Elle est désignée par un* identificateur *et peut contenir des méthodes.*

Nous verrons qu'une classe peut contenir d'autres éléments, comme par exemple des constantes (voir la section 3.6).

REMARQUE

Dans tout ce chapitre, nous parlerons de méthodes pour désigner en fait les méthodes de classe. Nous verrons au chapitre 7 qu'il existe une autre catégorie de méthodes, les méthodes d'instance. Pour l'instant, nous pouvons nous contenter des méthodes de classe.

3.1.2 Un exemple

Ecrivons un programme élémentaire :

Essai

```
1 public class Essai {
2   public static void main(String[] args) {
3     int i=2;
4   }
5 }
```

Nous venons d'écrire une classe et une méthode. Comme le mot clé `class` l'indique notre programme est en fait la définition de la classe `Essai`. Le contenu de la classe est l'ensemble des éléments définis entre l'accolade ouvrante qui suit l'identificateur de la classe (ligne 1) et l'accolade fermante qui lui correspond (ligne 5 du programme).

Notre classe contient ici une unique méthode, la méthode `main`. La ligne 2 débute la définition de la méthode, qui est constituée des instructions comprises entre l'accolade ouvrante qui termine la ligne 2 et l'accolade fermante qui lui correspond (ligne 4 du programme). Dans cet exemple, la méthode `main` ne comporte donc qu'une seule instruction. Le mot clé `void` de la ligne 2 indique que la méthode `main` n'a pas de résultat (voir la section 3.3.4 à ce sujet). Le texte `String[] args` décrit les paramètres de la méthode (voir la section 3.3.2).

REMARQUE

Le but ce chapitre n'est pas d'expliquer comment *écrire* des méthodes, mais comment *utiliser* des méthodes déjà existantes. Nous verrons au chapitre 6 comment créer nos propres méthodes, en dehors de la méthode `main` qui vient d'être étudiée.

Un programme `Java` comporte toujours une classe principale et une méthode `main` dans cette classe, c'est pourquoi nous avons appelé "programme" un fichier contenant une classe réduite à une méthode `main`.

3.1.3 Nom complet d'une méthode

Pour utiliser une méthode, il faut l'**appeler** : il s'agit d'indiquer au processeur qu'on souhaite qu'il exécute les instructions qui forment la méthode (voir la section suivante). Pour ce faire, il faut utiliser le **nom complet** de la méthode. Ce nom est obtenu en faisant suivre l'identificateur de la classe de la méthode par un point (i.e., le symbole "`.`") et par l'identificateur de la méthode. Il existe par exemple une classe `Math` qui contient entre autre une méthode `sin` qui est capable de calculer le sinus d'un réel. Son nom complet est donc `Math.sin`.

3.1.4 Conventions

Les noms de classes et de méthodes respectent des conventions, au même titre que les noms de variables :

- les noms de classes utilisent la même convention que les noms de programmes[1] (cf la section 1.3.4) ;
- les noms de méthodes utilisent la même convention que les noms de variables (cf la section 2.1.5).

3.2 Appel d'une méthode

3.2.1 Un exemple

Considérons le programme suivant :

```
double x=0;
double y;
y=Math.sin(x);
```

Le contenu de `y` après l'exécution de la troisième ligne du programme est 0, c'est-à-dire le sinus de 0. De ce fait, on peut considérer que la méthode `Math.sin` correspond à la version informatique de la fonction sinus. De plus, l'utilisation de la méthode est semblable à l'écriture mathématique. Pour indiquer que y correspond au sinus de x, on écrit en effet $y = \sin(x)$.

Il faut cependant se garder d'avoir une vision simpliste des méthodes en les considérant comme des fonctions mathématiques. Détaillons en effet le comportement du processeur pour l'exécution de la troisième ligne du programme :

1. comme pour toute affectation, le processeur procède en deux temps : il commence par évaluer la partie à droite du symbole `=`, puis il place la valeur résultat dans la variable à gauche du symbole `=` ;
2. le processeur doit donc évaluer l'expression `Math.sin(x)`, c'est-à-dire calculer le résultat de l'appel de la méthode. Il procède de la façon suivante :
   (a) il évalue les paramètres de la méthode. Dans notre exemple, il remplace `x` par sa valeur, à savoir 0 ;
   (b) il exécute les instructions qui constituent la méthode. L'une des instructions a pour but de définir le **résultat** de la méthode ;
   (c) la valeur de l'expression est par définition le résultat de la méthode.
3. la suite est classique : le processeur place le résultat de l'évaluation dans la variable `y`.

[1] Comme un programme est en fait une classe, c'est parfaitement logique.

3.2.2 Cas général

Appel d'une méthode

Dans le cas général, une méthode est appelée en écrivant son nom complet, suivi d'une paire de parenthèses. Ces parenthèses contiennent les paramètres éventuels de la méthode, séparés par des virgules. Les paramètres sont des expressions. Quand une méthode ne comporte pas de paramètres, on doit quand même faire suivre son nom d'une paire de parenthèses. On a donc la forme générale suivante pour un appel de méthode :

NomDeLaClasse.nomDeLaMéthode(*paramètre_1*,...,*paramètre_n*)

REMARQUE

Précisons encore une fois que la notation `...` n'a aucun sens en `Java`. Elle est utilisée ici pour indiquer que la méthode peut avoir aucun paramètre ou au contraire plusieurs.

Exécution de l'appel

Quand il rencontre un appel de méthode, le processeur l'exécute de la façon suivante :

1. il commence par évaluer les paramètres éventuels, de gauche à droite ;
2. il exécute les instructions qui constituent la méthode ;
3. si la méthode définit un résultat, celui-ci est considéré comme la **valeur de l'appel**. On dit alors que la méthode **renvoie un résultat**.

On doit donc noter qu'un appel de méthode peut être considéré comme l'évaluation de l'expression constituée par cet appel.

Exemples

La classe `Math` propose de nombreuses méthodes utiles (cf la section 3.4.2), comme par exemple `random`. Cette méthode choisit au hasard un nombre réel entre 0 et 1 (un `double`) et le définit comme son résultat. La méthode `random` n'utilise pas de paramètre. On peut écrire le programme suivant :

```
double y=Math.random();
```

Après l'exécution du programme, la variable `y` contient une valeur aléatoire. Cela signifie que si on exécute plusieurs fois le programme, la valeur contenue dans la variable `y` sera différente à chaque fois. On a seulement la garantie que cette valeur sera comprise entre 0 et 1. On peut obtenir 0.07489188941089064, puis 0.7533880380459403, etc.

Il faut bien noter qu'il est **obligatoire** d'indiquer des parenthèses après le nom complet de la méthode, même si celle-ci ne demande pas de paramètre, comme le montre l'exemple suivant :

Exemple 3.1 :

On considère le programme suivant :

MauvaisAppel

```
1 public class MauvaisAppel {
2   public static void main(String[] args) {
3     double y=Math.random;
4   }
5 }
```

Comme il manque la paire de parenthèses après `Math.random`, le compilateur refuse le programme, en donnant un message d'erreur assez peu parlant :

Erreur de compilation

```
MauvaisAppel.java:3: cannot resolve symbol
symbol  : variable random
location: class java.lang.Math
    double y=Math.random;
                  ^
1 error
```

En fait, le compilateur indique qu'il ne trouve pas de variable `random` dans la classe `Math`. En effet, c'est la présence des parenthèses qui permet au compilateur de comprendre qu'on souhaite faire un appel de méthode. Quand les parenthèses sont absentes, l'ordinateur cherche un autre élément dans la classe, en l'occurrence une variable (ou une constante, cf la section 3.6).

La classe `Math` est aussi capable, par exemple, de calculer le plus grand de deux nombres. Pour ce faire, on utilise la méthode `max`, comme dans le programme suivant :

```
double x=2,y=3;
double z=Math.max(x,y);
```

Bien entendu, l'exemple est ici très artificiel. Le contenu de `z` après l'appel est évidemment 3. Comme la méthode `max` demande deux paramètres, il est impossible de l'utiliser avec plus ou moins de paramètres, comme le montre l'exemple suivant :

Exemple 3.2 :

MauvaisParams

```
1  public class MauvaisParams {
2    public static void main(String[] args) {
3      double x=2,y=3,z=4;
4      double u=Math.max(x);
5      double v=Math.max(x,y,z);
6    }
7  }
```

Le compilateur détecte ici deux erreurs :

Erreur de compilation

```
MauvaisParams.java:4: cannot resolve symbol
symbol  : method max  (double)
location: class java.lang.Math
    double u=Math.max(x);
                  ^
MauvaisParams.java:5: cannot resolve symbol
symbol  : method max  (double,double,double)
location: class java.lang.Math
    double v=Math.max(x,y,z);
                  ^
2 errors
```

Encore une fois, les messages d'erreur ne sont pas faciles à interpréter de prime abord. En fait, le compilateur indique qu'il ne peut pas trouver (*cannot resolve symbol*) les méthodes qu'on tente d'utiliser. Il indique ensuite les méthodes que le programme tente d'utiliser d'après lui. Il indique par exemple `method max (double)`. Cela signifie pour le compilateur que le programme cherche à utiliser une méthode `max` demandant un paramètre de type `double` : cela correspond à l'appel de la ligne 4 qui ne comporte qu'un seul paramètre. De la même façon, le compilateur pense que le programme cherche à utiliser une méthode `max` demandant trois paramètres de type `double` à la ligne 5.

3.3 Typage d'un appel de méthode

3.3.1 Introduction

Nous avons pour l'instant évité un point délicat, à savoir la façon dont les types sont pris en compte par un appel de méthode. Il faut d'abord noter que les types interviennent à deux niveaux :

1. avant l'appel de la méthode, le processeur évalue les paramètres. Il faut que les valeurs obtenues soient compatibles avec ce qu'attend la méthode ;
2. quand la méthode définit un résultat, l'appel possède une valeur. Il faut que le type de cette valeur soit compatible avec l'utilisation qu'on souhaite en faire.

3.3.2 Signatures des méthodes et des appels de méthodes

Signature d'une méthode

Quand on définit une méthode, on définit indirectement sa **signature**. La signature d'une méthode est constituée de l'identificateur de celle-ci et de la liste des types des paramètres qu'elle demande pour fonctionner correctement. Voici quelques exemples :

- la méthode `random` de la classe `Math` ne prend pas de paramètre. De ce fait, sa signature est `random()` ;
- la méthode `sin` de la classe `Math` fonctionne avec un paramètre de type `double`. Sa signature est donc `sin(double)` ;
- il existe plusieurs méthodes `max` dans la classe `Math`, ce qui permet d'avoir une méthode adaptée à chaque type. Chaque méthode fonctionne avec deux paramètres du même type. On a donc les signatures `max(double,double)`, `max(int,int)`, etc.

REMARQUE

Dans un programme, la signature de la méthode `main` (cf la section 3.1.2) doit obligatoirement être `main(String[])`. Nous verrons dans les chapitres suivants comment interpréter cette signature.

Signature d'un appel de méthode

Quand on écrit un appel de méthode, on définit indirectement sa **signature**. La signature de l'appel est constitué de l'identificateur de la méthode appelée et de la liste des types des paramètres donnés dans l'appel. L'appel `Math.max(2,3.5,4)` correspond par exemple à la signature `max(int,double,int)`.

Considérons le programme suivant :

```
double x=2,y=3.5;
double z=Math.max(x,y);
```

La signature de l'appel est `max(double,double)`. En effet, comme toujours, le typage est *statique*. Il est effectué par le compilateur et ne tient pas compte des valeurs contenues par les variables, mais des types des variables seulement.

Compatibilité d'un appel de méthode et d'une méthode

Quand on souhaite utiliser une méthode, on effectue un appel. Or, la méthode et l'appel possèdent chacun une signature. Pour que le compilateur accepte l'appel, il faut que les **signatures soient compatibles**.

On dit que la signature d'un appel de méthode est compatible avec la signature d'une méthode si et seulement si :

1. les deux signatures comportent le même nombre de paramètres ;
2. chaque type de la signature de la méthode est plus général (au sens large) que le type correspondant de la signature de l'appel.

Le compilateur n'accepte un appel de méthode que si la signature de l'appel est compatible avec celle de la méthode qu'on souhaite appeler. On obtient ainsi une explication simple des erreurs indiquées dans l'exemple 3.2. En effet, les appels de la méthode `Max` ne comportent pas le bon nombre de paramètres. Donc, les signatures des appels ne sont pas compatibles avec la signature de la méthode `max` qui comporte toujours exactement deux paramètres (par exemple `max(double,double)`). D'ailleurs, le message d'erreur indique explicitement les signatures des appels, en précisant que le compilateur ne réussit pas à trouver les méthodes correspondantes.

La deuxième condition de compatibilité permet d'écrire simplement les appels de méthodes. C'est l'équivalent pour les méthodes de la règle de compatibilité des types (qui permet par exemple de placer un `int` dans une variable de type `double`). Voici un exemple d'utilisation de cette règle :

```
double x=Math.sin(1);
```

La signature de l'appel de méthode est `sin(int)`. Or, la signature de la méthode `sin` est `sin(double)`. Les deux signatures sont donc différentes. Cependant, cela ne pose aucun problème, car elles sont compatibles : on peut toujours placer un `int` dans un `double`.

REMARQUE

La règle de compatibilité peut sembler complexe. Il n'en est rien. En fait, une méthode attend des paramètres d'un type précis, dans un ordre fixé à l'avance. La règle de compatibilité demande simplement que les paramètres proposés lors de l'appel de la méthode soient compatibles avec ce que la méthode attend, c'est-à-dire en fait qu'ils soient d'un type moins général (au sens large) que le type attendu.

3.3.3 Résultat d'une méthode

Quand on définit une méthode, on indique le type de son éventuel résultat. Plus précisément, soit on indique `void`, ce qui signifie que la méthode n'a pas de résultat, soit on donne le type du résultat. La méthode `main` (cf la section 3.1.2) est toujours définie comme ne produisant pas de résultat, ce qui explique l'utilisation de `void` dans la ligne qui comporte l'identificateur `main`.

Quand une méthode définit un résultat, c'est-à-dire quand elle renvoie une valeur, ce résultat est considéré comme la valeur de l'appel de la méthode. Son type est déterminé **de façon statique** par la définition de la méthode. Il ne dépend en aucun cas des valeurs des paramètres, mais seulement de leur types. Pour que l'utilisation d'une méthode soit considérée comme correcte, il faut bien

entendu que le type de son résultat soit compatible avec l'utilisation qu'en fait le programme, comme l'illustre l'exemple suivant :

Exemple 3.3 :

La classe `Math` propose une méthode `sqrt`, de signature `sqrt(double)`, qui calcule la racine carrée de son paramètre. Son résultat est de type `double`. De ce fait, le programme suivant est incorrect :

MauvaisSqrt

```
1 public class MauvaisSqrt {
2   public static void main(String[] args) {
3     int i=4;
4     int j=Math.sqrt(4);
5   }
6 }
```

Le compilateur refuse le progamme et donne le message suivant :

ERREUR DE COMPILATION

```
MauvaisSqrt.java:4: possible loss of precision
found   : double
required: int
    int j=Math.sqrt(4);
                   ^
1 error
```

Il faut d'abord bien noter que le problème ne vient pas de l'appel de méthode en lui-même. En effet, la signature de l'appel est `sqrt(int)`, qui est compatible avec `sqrt(double)`. C'est l'utilisation de la valeur de l'appel qui pose problème : la méthode renvoie toujours un `double` et on ne peut pas placer une valeur de ce type dans une variable de type `int`.

3.3.4 Les méthodes sans résultat

Quand une méthode ne renvoie pas de résultat, il est **impossible** de l'utiliser dans une expression. L'appel d'une méthode sans résultat ne possède en effet pas de valeur. L'intérêt d'une méthode sans valeur est qu'elle produit un effet "extérieur" : elle peut provoquer par exemple un affichage à l'écran (cf la section 3.5.2).

Pour l'instant, nous nous contenterons de l'exemple de la méthode `exit` de la classe `System`. Cette méthode possède la signature `exit(int)`. Elle a pour effet de provoquer l'arrêt du programme, même s'il reste des instructions après elle. Le paramètre entier qu'elle utilise permet de transmettre un code de sortie pour le programme. Un code différent de 0 indique une erreur. Voici un exemple d'utilisation :

```
int i=2;
System.exit(0);
int j=3;
```

La troisième ligne de ce programme ne sera jamais exécutée, car l'appel de la méthode en ligne 2 provoque l'arrêt du programme. L'exemple suivant montre ce qui se passe quand on tente d'utiliser une méthode sans résultat comme si elle possédait une valeur.

Exemple 3.4 :

On considère le programme suivant :

PasDeValeur

```
1 public class PasDeValeur {
2   public static void main(String[] args) {
3     int k=System.exit(0);
4   }
5 }
```

Le compilateur refuse ce programme et donne le message d'erreur suivant :

ERREUR DE COMPILATION

```
PasDeValeur.java:3: incompatible types
found   : void
required: int
    int k=System.exit(0);
                      ^
1 error
```

On voit constate que le message n'est pas forcément très clair. Il faut impérativement savoir que `void` correspond au cas d'une méthode sans résultat.

REMARQUE

Contrairement à ce qu'on pourrait penser, il n'est pas obligatoire d'utiliser le résultat d'une méthode qui en propose un. Le programme suivant est donc correct :

SansUtilisation

```
1 public class SansUtilisation {
2   public static void main(String[] args) {
3     Math.sin(2.5);
4   }
5 }
```

Bien entendu, ce programme est totalement inutile. Mais il arrive qu'une méthode possède un résultat et provoque en même temps un effet "extérieur". On peut alors avoir seulement besoin de l'effet extérieur, sans se soucier du résultat.

3.4 Quelques méthodes de calcul

3.4.1 Convention de présentation

Pendant notre apprentissage, nous allons être amenés à utiliser de nombreuses méthodes déjà définies en `Java`. Cet ouvrage comportera donc régulièrement des sections de documentation[2] donnant des listes de méthodes utiles. Pour pouvoir utiliser une méthode, il faut bien sûr connaître sa classe et son identificateur, mais il faut aussi savoir quels paramètres elle demande et quel résultat elle produit. Pour chaque méthode, nous indiquerons sa signature précédée du type du résultat qu'elle produit (éventuellement `void` pour une méthode sans résultat). Puis nous décrirons l'effet de la méthode. La méthode `sin` de la classe `Math` sera donc par exemple décrite commme suit :

`double sin(double)`

Calcule et renvoie le sinus de son paramètre.

[2] Pour une documentation complète, on se reportera à [12].

3.4.2 La classe `Math`

La classe `Math` propose de nombreuses méthodes permettant de réaliser des calculs scientifiques élémentaires. Voici les méthodes les plus utiles :

`double abs(double)`
: Renvoie la valeur absolue de son paramètre.

`float abs(float)`
: Cf méthode précédente.

`long abs(long)`
: Cf méthode précédente.

`int abs(int)`
: Cf méthode précédente.

`double acos(double)`
: Renvoie l'arc cosinus du paramètre (le résultat est élément de $[0, \pi]$).

`double asin(double)`
: Renvoie l'arc sinus du paramètre (le résultat est élément de $[-\frac{\pi}{2}, \frac{\pi}{2}]$)

`double atan(double)`
: Renvoie l'arc tangente du paramètre (le résultat est élément de $[-\frac{\pi}{2}, \frac{\pi}{2}]$[3])

`double ceil(double)`
: Renvoie l'arrondi supérieur du paramètre. Il s'agit de la plus petite valeur entière supérieure ou égale au paramètre de la méthode. Il faut noter que cette méthode renvoie un `double`, même si les valeurs obtenues sont toujours entières.

`double cos(double)`
: Renvoie le cosinus du paramètre (qui doit être exprimé en radians).

`double exp(double)`
: Renvoie l'exponentielle du paramètre.

`double floor(double)`
: Renvoie la partie entière du paramètre, c'est-à-dire la plus grande valeur entière inférieure ou égale au paramètre de la méthode. Il faut noter que cette méthode renvoie un `double`, même si les valeurs obtenues sont toujours entières.

`double log(double)`
: Renvoie le logarithme népérien du paramètre.

`double max(double,double)`
: Renvoie le plus grand des deux paramètres.

`float max(float,float)`
: Cf méthode précédente.

`long max(long,long)`
: Cf méthode précédente.

`int max(int,int)`
: Cf méthode précédente.

[3] Comme le processeur manipule les "valeurs" $+\infty$ et $-\infty$, il leur associe naturellement les limites associées pour certaines fonctions, comme par exemple pour l'arc tangente.

`double min(double,double)`
: Renvoie le plus petit des deux paramètres.

`float min(float,float)`
: Cf méthode précédente.

`long min(long,long)`
: Cf méthode précédente.

`int min(int,int)`
: Cf méthode précédente.

`double pow(double,double)`
: Renvoie la valeur du premier paramètre mis à la puissance du second paramètre. La valeur de `Math.pow(x,y)` est donc x^y.

`double random()`
: Renvoie un `double` choisit au hasard dans l'intervalle $[0, 1]$.

`double rint(double)`
: Renvoie l'arrondi du paramètre, c'est-à-dire la valeur entière la plus proche de celui-ci. Il faut noter que cette méthode renvoie un `double`, même si les valeurs obtenues sont toujours entières.

`long round(double)`
: Renvoie l'entier `long` le plus proche du paramètre.

`int round(float)`
: Renvoie l'entier `int` le plus proche du paramètre[4].

`double sin(double)`
: Renvoie le sinus du paramètre (qui doit être exprimé en radians).

`double sqrt(double)`
: Renvoie la racine carrée du paramètre.

`double tan(double)`
: Renvoie la tangente du paramètre (qui doit être exprimé en radians).

REMARQUE

On remarque que dans la classe `Math`, plusieurs méthodes possèdent le même identificateur. On appelle ce phénomène une **surchage de méthode**. C'est le cas par exemple de l'identificateur `abs` qui correspond à quatre méthodes différentes. Ceci est possible car chacune des méthodes possède une signature unique. Comme le compilateur se base sur la signature pour déterminer la méthode appelée, il n'y a pas de risque de confusion. Si on veut par exemple calculer la valeur absolue d'un entier `int`, on peut utiliser `Math.abs`. Le compilateur choisit automatiquement la méthode de signature `abs(int)`. De cette manière, le résultat est de type `int`, ce qui permet son utilisation avec une variable de type `int`, par exemple. Le programme suivant est donc parfaitement correct :

```
int i=-3;
int j=Math.abs(i);
```

[4] Notez bien que le paramètre est de type `float`.

3.4.3 Les classes `Double` et `Float`

Aux types fondamentaux `double` et `float` sont associés deux classes portant presque le même nom. Ces classes proposent des méthodes utiles pour vérifier les résultats d'un calcul. Commençons par la classe `Double` :

`boolean isInfinite(double)`
: Renvoie `true` si et seulement si le paramètre correspond au code qui désigne une valeur infinie, positive ou négative.

`boolean isNaN(double)`
: Renvoie `true` si et seulement si le paramètre correspond au code qui désigne une valeur indéterminée, comme par exemple le résultat de la division de zéro par lui-même.

La classe `Float` propose les mêmes méthodes, adaptées pour les `float`s, ce qui donne :

`boolean isInfinite(float)`
: Renvoie `true` si et seulement si le paramètre correspond au code qui désigne une valeur infinie, positive ou négative.

`boolean isNaN(float)`
: Renvoie `true` si et seulement si le paramètre correspond au code qui désigne une valeur indéterminée.

REMARQUE

Que se passe-t-il quand on tente de faire faire par une méthode un calcul impossible, comme par exemple calculer la racine carrée d'un nombre négatif ? Comme nous l'avons indiqué à la section 2.4.1, il n'y a pas de calcul impossible pour les réels. Il est donc logique que les méthodes numériques respectent cette règle. De ce fait, aucune méthode de la classe `Math` ne risque de provoquer d'erreur d'exécution. Par contre, les résultats obtenus seront parfois inutilisables car non définis. Considérons par exemple :

```
boolean b=Double.isNaN(Math.sqrt(-1.5));
```

Après l'exécution du programme, `b` contient la valeur `true`, le résultat de `Math.sqrt(-1.5)` étant le code spécial réservé à la valeur indéfinie.

3.5 Entrées et sorties

3.5.1 Introduction et vocabulaire

Les programmes que nous avons écrits jusqu'à présent sont singulièrement limités. En effet, ils ne peuvent pas interagir avec l'utilisateur. Celui-ci ne peut pas saisir des informations et l'ordinateur ne peut pas afficher de résultat. Dans cette section, nous présentons brièvement les instructions qui permettent d'interagir avec l'utilisateur.

Il est important de comprendre que l'interaction avec l'utilisateur est un sujet délicat techniquement. En effet, comme nous l'avons déjà évoqué, l'ordinateur **représente** dans la mémoire des valeurs en utilisant comme support des chiffres binaires. Le *code* qui est utilisé est assez éloigné de la façon dont nous écrivons un nombre sur une feuille de papier[5]. De ce fait, la saisie d'un nombre et son affichage seront des opérations de *codage* et *décodage*.

On appelle **entrée** une action de *saisie* par l'utilisateur d'une valeur : l'utilisateur "entre" une valeur "dans" l'ordinateur. On appelle **sortie** un affichage d'une valeur à l'écran de l'ordinateur : il "sort" une valeur pour la montrer à l'utilisateur.

[5] En fait, comme nous l'avons vu à la section 2.4.2, il s'agit surtout d'une représentation en base deux.

3.5.2 Affichage d'une valeur

L'affichage d'une valeur est relativement simple, comme le montre l'exemple suivant :

Exemple 3.5 :

On considère le programme suivant :

AffichageValeur

```
1  public class AffichageValeur {
2    public static void main(String[] args) {
3      System.out.println(2);
4      System.out.println(2.5*3.5);
5      double x=3;
6      System.out.println(x);
7      double y=6.5;
8      System.out.println(2*(x-y));
9      System.out.println(Math.sqrt(y-0.5*x));
10   }
11 }
```

Quand on l'exécute, il produit l'affichage suivant :

AFFICHAGE

```
2
8.75
3.0
-7.0
2.23606797749979
```

On dévine donc que chaque utilisation de `System.out.println` provoque l'affichage de la valeur de l'expression entre parenthèses.

De façon plus formelle, on dispose en fait de deux méthodes, `print` et `println` qui provoquent l'affichage de leur unique paramètre. Ces deux méthodes ne produisent pas de résultat (`void`). Elles sont d'une nature un peu particulière, sur laquelle nous reviendrons au chapitre 7. En fait, la classe `System` contient une constante (cf la section 3.6), nommée `out`. Pour se servir de cette constante, il suffit d'utiliser son nom complet, à savoir `System.out`. Le contenu de la constante est une référence vers un **objet**. Pour l'instant, nous admettrons qu'en faisant suivre le nom complet de la constante par le nom d'une méthode (adaptée), nous obtenons un appel de méthode, en tout point semblable à ceux que nous avons étudiés dans le début de ce chapitre. Nous utiliserons donc les deux appels de méthode suivant :

```
System.out.print(expression);
System.out.println(expression);
```

La première méthode permet d'afficher la valeur de l'expression entre parenthèses *sans passer à la ligne après cet affichage*. La deuxième méthode passe au contraire à la ligne après l'affichage. On peut aussi utiliser la méthode `void println()`. Cette méthode ne prend pas de paramètre et a pour effet d'afficher une ligne vide, soit en fait de faire passer à la ligne.

Voici un nouvel exemple d'utilisation :

Exemple 3.6 :

On considère le programme suivant :

NouvelAffichageValeur

```
1  public class NouvelAffichageValeur {
2    public static void main(String[] args) {
3      char b='b';
4      System.out.print(b);
5      System.out.print('a');
6      System.out.print(b);
7      System.out.println('a');
8      System.out.println();
9      double x=Math.random();
10     double y=Math.random();
11     System.out.println(x);
12     System.out.println(y);
13     System.out.println(x>y);
14   }
15 }
```

Quand on l'exécute, il produit l'affichage suivant (par exemple) :

AFFICHAGE

```
baba

0.958921857381109
0.7473913064047328
true
```

On voit bien l'effet du `print` comparé à `println` sur la première ligne de l'affichage, ainsi que l'effet du `println` sans paramètre. On remarque qu'on peut bien entendu afficher des `boolean`s ou des `char`s.

Notons pour finir que les méthodes `print` et `println` sont capables d'afficher les valeurs spéciales associées aux `double`s et `float`s, comme l'illustre l'exemple suivant :

Exemple 3.7 :

Le programme suivant affiche toutes les valeurs spéciales :

AffichageFloatEtDouble

```
1  public class AffichageFloatEtDouble {
2    public static void main(String[] args) {
3      // valeur infinie positive
4      System.out.println(1.0/0.0);
5      // valeur infinie négative
6      System.out.println(-1.0/0.0);
7      // valeurs indéterminées
8      System.out.println(0.0/0.0);
9      System.out.println((1.0/0.0)-(1.0/0.0));
10     // la même chose en float
11     System.out.println(1.0f/0.0f);
```

```
12      System.out.println(-1.0f/0.0f);
13      System.out.println(0.0f/0.0f);
14      System.out.println((1.0f/0.0f)-(1.0f/0.0f));
15    }
16  }
```

Il produit l'affichage suivant :

AFFICHAGE

```
Infinity
-Infinity
NaN
NaN
Infinity
-Infinity
NaN
NaN
```

On constate qu'il n'existe aucune distinction entre les `float`s et les `double`s pour ces affichages. C'est d'ailleurs le cas en général.

3.5.3 Affichage de texte

Les affichages proposés dans la section précédente sont assez limités. Ils ne permettent pas en effet d'afficher autre chose que des valeurs booléennes ou numériques, ainsi que des caractères seuls. Il est donc impossible de donner à l'utilisateur une explication concernant la valeur qui est affichée, ce qui rend le programme difficile à utiliser.

Fort heureusement, les méthodes `print` et `println` peuvent aussi accepter comme paramètres un texte à afficher :

Exemple 3.8 :

On considère le programme suivant :

CalculMoyenne

```
1  public class CalculMoyenne {
2    public static void main(String argc[]) {
3      float x=17.5f,y=12.3f;
4      System.out.print("Moyenne : ");
5      System.out.println((x+y)/2);
6    }
7  }
```

Ce programme affiche le texte suivant :

AFFICHAGE

```
Moyenne : 14.9
```

La règle à suivre est relativement simple. Le paramètre de la méthode d'affichage utilisée (`print` ou `println`) est un texte compris entre des guillemets (`"un texte"`). L'ordinateur affiche à l'écran le texte **tel quel**. Ceci signifie qu'un texte contenant une expression par exemple sera affiché sans que l'expression soit évaluée. L'appel `System.out.println("(x+y)/2")` affiche à l'écran `(x+y)/2`

et ne doit surtout pas être confondu avec `System.out.println((x+y)/2)` qui affiche le résultat d'un calcul.

Nous remarquons que ce système d'affichage devient rapidement lassant. En effet, pour afficher `x=5`, où `x` est le nom d'une variable et `5` la valeur qu'elle contient, nous serons obligés d'écrire :

```
int x=5;
System.out.print("x=");
System.out.println(x);
```

Nous devons donc écrire relativement souvent du texte très répétitif. Fort heureusement, `Java` permet d'éviter ce genre de répétition. Pour ce faire, on utilise l'opérateur d'addition (`+`) pour symboliser la mise bout à bout[6] d'éléments à afficher. Le morceau de programme ci-dessus devient alors :

```
int x=5;
System.out.println("x="+x);
```

Pour bien comprendre l'affichage, il suffit de considérer que les symboles `+` qui apparaissent dans une expression paramètre d'une méthode d'affichage permettent de fabriquer un texte, en concaténant les textes qui leur servent d'opérandes. Dans l'exemple qui précède, le texte à afficher est fabriqué en ajoutant le contenu de la variable `x` (c'est-à-dire `5`) à la fin du texte `"x="`, ce qui donne donc `"x=5"`.

Le seul point délicat relatif à cette utilisation est que le compilateur applique au `+` de concaténation les mêmes règles de priorité qu'au `+` de l'addition. Etudions un exemple :

Exemple 3.9 :

```
int x=5,y=-5;
System.out.println("x+y="+x+y);
System.out.println("(x+y)="+(x+y));
```

Ce programme n'affiche pas :

```
x+y=0
(x+y)=0
```

mais :

Affichage

```
x+y=5-5
(x+y)=0
```

En effet, dans le premier affichage, à cause des règles de priorité, l'ordinateur abstrait cherche à afficher `("x+y="+x)+y`. En effectuant le premier "calcul" (c'est-à-dire la concaténation), il réduit cette expression à `"x+y=5"+y`, puis à `"x+y=5-5"`. Dans le deuxième affichage au contraire, la présence de parenthèses fait que l'addition est bien réalisée avant la concaténation, ce qui donne un résultat plus conforme à ce qu'on attendait.

[6] Le terme technique est concaténation.

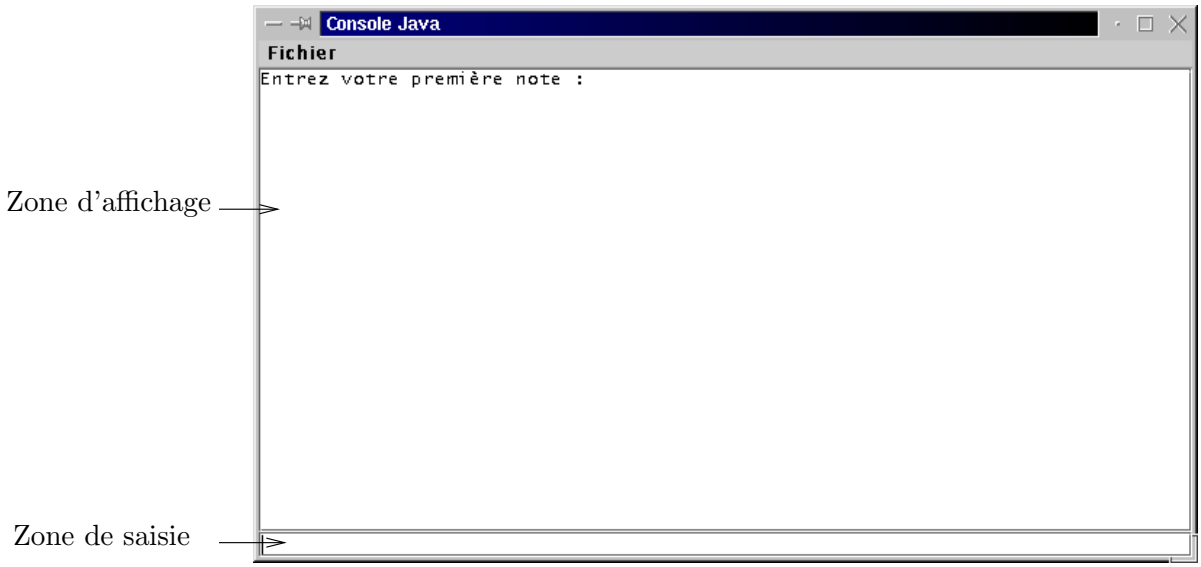

FIG. 3.1 – La console après ouverture et affichage

3.5.4 Les saisies

Introduction

La saisie d'une information par l'utilisateur est une opération plus complexe que l'affichage. En effet, pour l'affichage, aucune erreur ne peut se produire, car le processeur sait comment l'information est représentée dans la mémoire et peut donc convertir un nombre en une suite de caractères qu'il transmet à l'écran.

Pour la saisie, le problème est moins simple. L'utilisateur est libre de taper ce qu'il souhaite au clavier. Le processeur reçoit donc une suite de caractères et doit chercher à les convertir en un nombre. Si l'utilisateur tape `43.3` et que le processeur s'attend à lire un entier, il doit indiquer que la saisie est erronée.

En `Java`, la réponse à ce problème est très complexe. C'est pourquoi j'ai écrit un ensemble de méthodes permettant de simplifier les saisies. Ces méthodes restent cependant complexes et nous ne pourrons expliquer complètement leur utilisation que dans la suite du cours. Il est important de noter que ces méthodes sont **spécifiques** à Dauphine[7] et ne font pas partie de `Java`. Pour pouvoir les utiliser, il faut **impérativement** ajouter comme **première** ligne du programme la ligne `import dauphine.util.*;`. Cette ligne indique au compilateur qu'il peut utiliser une nouvelle classe (la classe `Console`) définie à Dauphine[8]. Nous reviendrons sur le sens de cette ligne dans la section 3.7.

Un exemple d'utilisation

Exemple 3.10 :

Voici un nouveau programme de calcul de moyenne :

CalculMoyenneSaisie

```
1 import dauphine.util.*;
2 public class CalculMoyenneSaisie {
```

[7] Le fichier `jar` qui permet l'utilisation de ces méthodes est accessibles à l'URL `ftp://ftp.ufrmd.dauphine.fr/pub/java/dauphine/dauphine.jar`. Ce fichier ne fonctionne qu'avec Java 2 (versions 1.2 et 1.3).

[8] Il faut bien entendu que le compilateur soit capable de trouver cette nouvelle classe, ce qui veut dire qu'il doit être bien configuré, point qui dépasse le cadre de cet ouvrage.

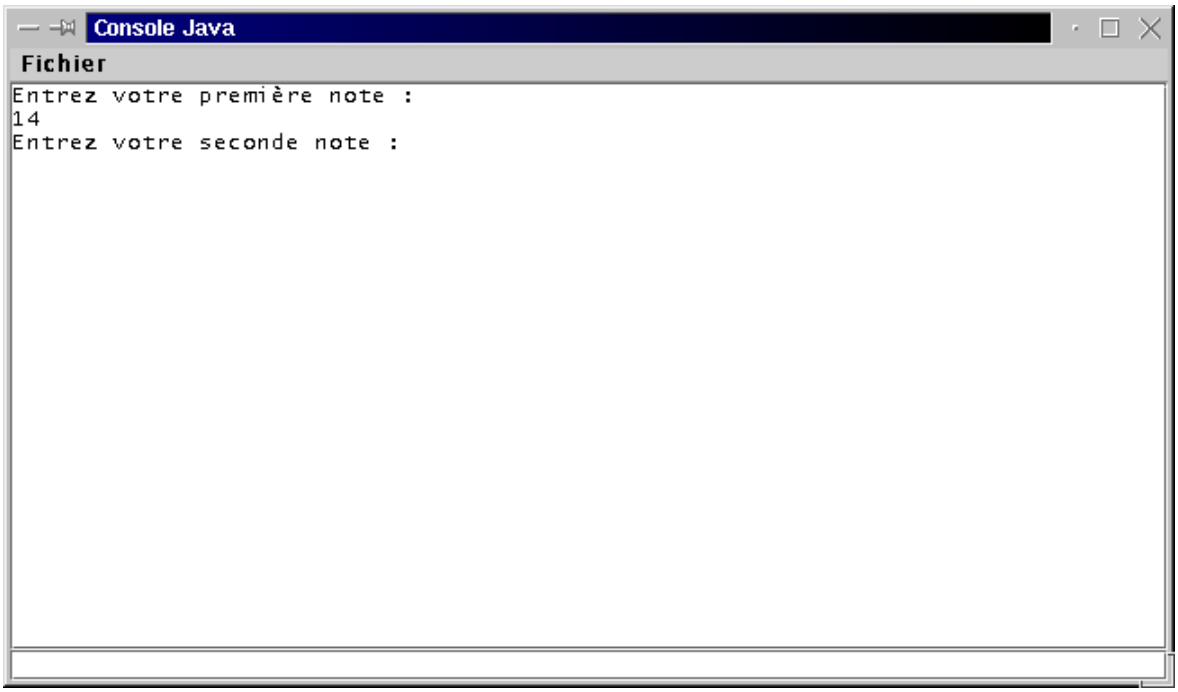

FIG. 3.2 – La console après une saisie

```
3     public static void main(String argc[]) {
4       Console.start();
5       int a,b;
6       System.out.println("Entrez votre première note : ");
7       a = Console.readInt();
8       System.out.println("Entrez votre seconde note : ");
9       b = Console.readInt();
10      System.out.println("Votre moyenne est "+(a+b)/2);
11    }
12  }
```

Quand on exécute ce programme, le résultat est complètement nouveau pour nous. En effet, une fenêtre apparaît sur l'écran de l'ordinateur. Le texte du premier affichage demandé (ligne 6) apparaît dans la zone principale (la zone d'affichage) de la fenêtre (voir la figure 3.1). A ce moment, l'ordinateur attend que l'utilisateur saisisse quelque chose dans la zone de saisie de la fenêtre (le rectangle du bas de la fenêtre). Supposons par exemple que l'utilisateur saisisse la valeur `14`. L'affichage est alors modifié. La valeur saisie s'affiche sur la deuxième ligne, puis l'affichage demandé à la ligne 8 du programme se produit. L'affichage obtenu est donné par la figure 3.2. L'ordinateur attend une nouvelle saisie. Supposons maintenant que l'utilisateur saisisse `17`. Le dernier affichage prévu dans le programme (ligne 10) s'effectue tel que l'illustre la figure 3.3. Grâce au dernier affichage, on devine que la méthode `readInt` de la classe `Console` permet d'attendre la saisie par l'utilisateur d'un entier dans la zone de saisie de la fenêtre.

Démarrage

Les saisies sont donc réalisées par l'intermédiaire de la classe `Console`. Tout programme qui souhaite utiliser cette classe doit commencer par un appel à la méthode `start` :

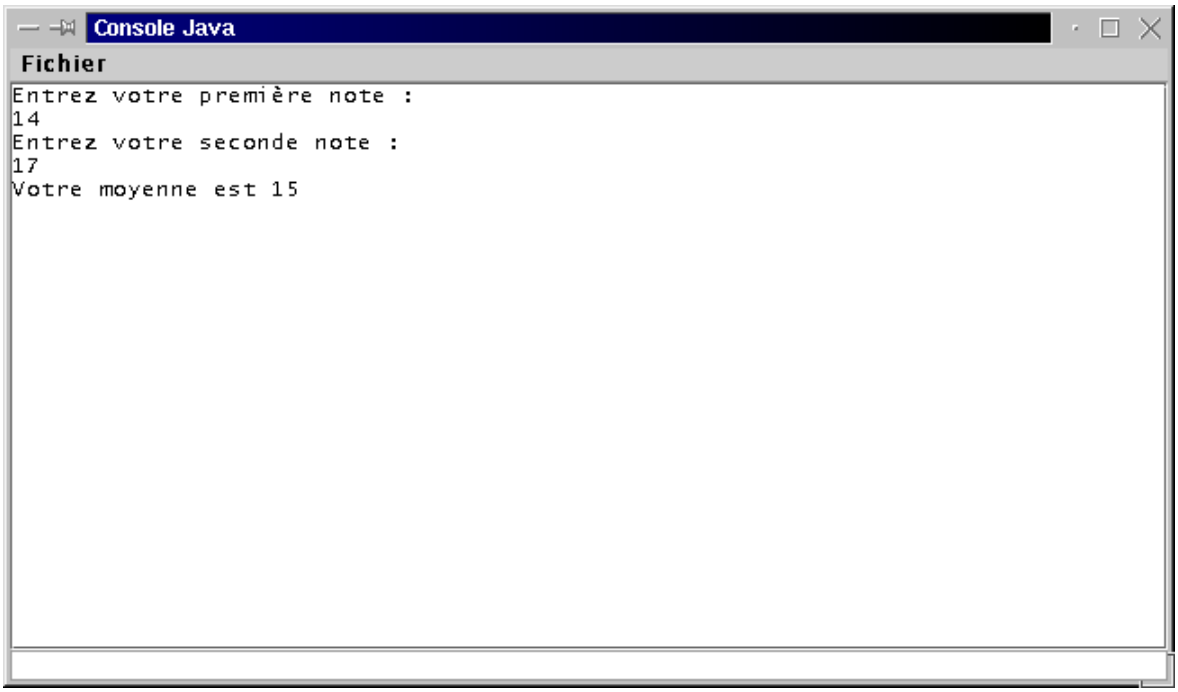

FIG. 3.3 – La console après les deux saisies

`void start()`

Démarre la fenêtre de saisie et autorise l'utilisation des méthodes de lecture dans la suite du programme.

Comme l'indique la documentation, il n'est pas nécessaire que l'appel à cette méthode soit effectué comme première instruction du programme. Par contre, il faut **impérativement** que l'appel soit fait avant tout appel à une méthode de lecture, comme l'illustre l'exemple suivant :

Exemple 3.11 :

Considérons le programme suivant :

PasDeStart

```
1 import dauphine.util.*;
2 public class PasDeStart {
3   public static void main(String argc[]) {
4     int k=Console.readInt();
5     Console.start();
6     int j=Console.readInt();
7   }
8 }
```

Ce programme est accepté par le compilateur. Par contre, quand on l'exécute, il produit l'erreur suivante :

ERREUR D'EXÉCUTION

```
Exception in thread "main" dauphine.util.NoConsoleException: Console.start()
doit être appelée avant toute utilisation de la méthode readInt().
        at dauphine.util.Console.check(Console.java:19)
```

```
        at dauphine.util.Console.readInt(Console.java:34)
        at PasDeStart.main(PasDeStart.java:4)
```

Pour une fois le message est explicite (car en français !). Comme nous avons appelé `readInt` à la ligne 4 avant d'appeler `start` à la ligne suivante, le programme ne peut pas fonctionner.

Insistons sur le fait qu'il est absolument impératif d'indiquer la ligne `import dauphine.util.*;` au début du programme. Sans cette ligne, le programme ne compile pas, comme l'illustre l'exemple suivant :

Exemple 3.12 :

Dans le programme suivant, la ligne `import dauphine.util.*;` manque :

PasDeImport

```
1 public class PasDeImport {
2   public static void main(String argc[]) {
3     Console.start();
4     int i=Console.readInt();
5     System.out.println(i);
6   }
7 }
```

Le compilateur refuse le programme et donne le message suivant :

Erreur de compilation

```
PasDeImport.java:3: cannot resolve symbol
symbol  : variable Console
location: class PasDeImport
    Console.start();
    ^
PasDeImport.java:4: cannot resolve symbol
symbol  : variable Console
location: class PasDeImport
    int i=Console.readInt();
          ^
2 errors
```

Le compilateur indique donc qu'il ne reconnaît pas la classe `Console`.

Saisie

Pour faire une saisie, il suffit d'utiliser une des méthodes `readXxxx` de la classe `Console`, où `Xxxx` désigne le type de la valeur qu'on souhaite lire. Il faut bien sûr que la méthode `start` ait été appelée avant. Chaque méthode de saisie demande au processeur d'attendre que l'utilisateur tape quelque chose dans la zone de saisie de la fenêtre. Quand l'utilisateur valide sa saisie (par la touche "Entrée"), la méthode vérifie que le texte saisi correspond bien au type demandé. Si c'est le cas, elle renvoie la valeur saisie. Sinon, elle affiche un message d'erreur et attend une nouvelle saisie.

Voici la liste des méthodes utilisables :

`byte readByte()`

Lecture d'un `byte` : la méthode n'accepte que les valeurs entières comprises au sens large entre -128 et 127.

`short readShort()`
: Lecture d'un `short` : la méthode n'accepte que les valeurs entières comprises au sens large entre -32768 et 32767.

`int readInt()`
: Lecture d'un `int` : la méthode n'accepte que les valeurs entières comprises au sens large entre -2^{31} et $2^{31}-1$.

`long readLong()`
: Lecture d'un `long` : la méthode n'accepte que les valeurs entières comprises au sens large entre -2^{63} et $2^{63}-1$.

`float readFloat()`
: Lecture d'un `float` : la méthode accepte n'importe quelle valeur entière ou réelle, éventuellement en notation scientifique. La valeur est ensuite tronquée pour avoir 8 chiffres significatifs au maximum. Si la valeur est trop grande, le processeur utilise la valeur spéciale infinie.

`double readDouble()`
: Lecture d'un `double` : la méthode accepte n'importe quelle valeur entière ou réelle, éventuellement en notation scientifique. La valeur est ensuite tronquée pour avoir 16 chiffres significatifs au maximum. Si la valeur est trop grande, le processeur utilise la valeur spéciale infinie.

`boolean readBoolean()`
: Lecture d'un `boolean` : la méthode accepte les textes `true` et `false`, en majuscules comme en minuscules.

`char readChar()`
: Lecture d'un `char` : la méthode accepte n'importe quel caractère.

Affichage

Pour éviter d'alourdir cet ouvrage, nous ne donnerons plus la représentation graphique de la fenêtre ouverte par la méthode `start` de la classe `Console`. Nous nous contenterons de donner le contenu de la zone d'affichage de la fenêtre. L'affichage complet de l'exemple 3.10 sera donc donné de la façon suivante :

AFFICHAGE

```
Entrez votre première note :
14
Entrez votre seconde note :
17
Votre moyenne est 15
```

3.6 Les constantes de classe

3.6.1 Principe

Comme nous l'avons indiqué à la section 3.1.1, une classe est un groupe d'éléments `Java`. Nous avons vu qu'une classe peut ainsi contenir des méthodes. Elle peut aussi contenir des **constantes**. Une constante est une valeur non modifiable, désignée par un identificateur. Comme une variable, elle est stockée dans la mémoire de l'ordinateur, mais le compilateur interdit toute tentative de modification.

Pour se servir d'une constante, il suffit d'utiliser son nom complet, qui, comme pour les méthodes, est constitué du nom de la classe, suivi d'un point, suivi du nom de la constante. La classe `Math` définit par exemple la constante `PI`, correspondant au nombre réel π. Voici un exemple d'utilisation de cette constante :

Exemple 3.13 :

On considère le programme suivant :

DemoPi

```
1  public class DemoPi {
2    public static void main(String[] args) {
3      System.out.println(Math.PI);
4      System.out.println(Math.sin(Math.PI));
5      System.out.println(Math.sin(Math.PI/2));
6    }
7  }
```

L'affichage produit est le suivant :

AFFICHAGE

```
3.141592653589793
1.2246063538223773E-16
1.0
```

La première ligne donne la valeur de `PI`, qui est bien une approximation de π. La seconde valeur est une approximation de $\sin \pi$. On a ici une démonstration éloquente du fait que l'ordinateur donne des résultats imprécis (car on devrait obtenir 0). Enfin, la dernière ligne donne la valeur de $\sin \frac{\pi}{2}$ (cette valeur est parfaitement exacte, mais c'est un phénomène rare).

Comme une variable, une constante possède un type. Elle ne peut être utilisée qu'en respectant les types.

Si on tente de modifier une constante, le compilateur refuse le programme, comme l'illustre l'exemple suivant :

Exemple 3.14 :

Voici une tentative de modification de la constante `PI` :

TentativeDeModification

```
1  public class TentativeDeModification {
2    public static void main(String[] args) {
3      Math.PI=3.14;
4    }
5  }
```

Le compilateur refuse ce programme et affiche le message suivant :

ERREUR DE COMPILATION

```
TentativeDeModification.java:3: cannot assign a value to final variable PI
    Math.PI=3.14;
        ^
1 error
```

Le message d'erreur n'est pas très clair, essentiellement pour des raisons de vocabulaire. En effet, pour le compilateur `Java`, une constante est une *final variable*, c'est-à-dire une variable non modifiable. Le compilateur indique donc qu'on ne peut pas donner une valeur (*cannot assign a value*) à une constante.

3.6.2 Conventions

Les noms de constantes obéissent à des conventions particulières, de sorte qu'on évite de les confondre avec des variables. Un identificateur de constante vérifie les conventions suivantes[9] :

- toutes les lettres de l'identificateur sont des majuscules ;
- les mots qui forment l'identificateur sont séparés par le symbole souligné `_`.

Les sections suivantes contiennent de nombreux exemples d'application de ces conventions.

3.6.3 Quelques constantes utiles

Pour documenter les constantes, nous utiliserons une présentation similaire à celle utilisée pour les méthodes. Nous indiquerons le type de la constante, suivi de son nom, suivi d'une description de celle-ci. Voici par exemple une description de la constante `PI` :

`double PI`

Une valeur approchée de π.

La classe `Math`

La classe `Math` propose la constante `PI` que nous venons de documenter. Elle propose une autre constante :

`double E`

Une valeur approchée de e, c'est-à-dire du réel tel que $\ln e = 1$.

Les classes de traitement des réels

Comme nous l'avons vu à la section 3.4.3, il existe deux classes, `Double` et `Float` qui facilitent la manipulation des réels. Ces classes contiennent des constantes utiles. Pour la classe `Double`, nous avons les constantes suivantes :

`double MAX_VALUE`

La plus grande valeur positive représentable par un `double`. Comme nous l'avons indiqué au chapitre précédent, il s'agit du réel $1.7976931348623157\ 10^{308}$.

`double MIN_VALUE`

La plus petite valeur positive représentable par un `double`. Il s'agit de $4.9\ 10^{-324}$.

`double NaN`

La valeur spéciale indéterminée.

`double NEGATIVE_INFINITY`

La valeur spéciale représentant l'infini positif.

`double POSITIVE_INFINITY`

La valeur spéciale représentant l'infini négatif.

Pour la classe `Float`, nous avons exactement les mêmes constantes, avec bien entendu un codage différent (en `float`). Nous ne donnerons donc pas la description des constantes. Notons simplement que la valeur de `Float.MIN_VALUE` est $1.4\ 10^{-45}$, et que celle de `Float.MAX_VALUE` est $3.4028235\ 10^{38}$.

[9] Pour des raisons pratiques, ce n'est pas le cas la constante `out` de la classe `System`.

Les classes de traitement des entiers

Pour chaque type entier, il existe une classe correspondante, comme l'indique le tableau suivant :

type	classe
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`char`	`Character`

Chaque classe contient (entre autre) deux constantes, `MAX_VALUE` et `MIN_VALUE`. La constante `MAX_VALUE` correspond à la plus grande valeur représentable par le type considéré, alors que `MIN_VALUE` correspond à la plus petite valeur. Contrairement aux réels, il ne s'agit pas de la valeur absolue, mais bien de la valeur minimale, donc négative (sauf pour le cas des `char`s). La valeur de `Short.MIN_VALUE` est donc -32768, codé comme un `short`.

3.7 Les paquets

3.7.1 Motivation

Quand on développe des programmes professionnels, l'organisation des méthodes en classes n'est pas suffisante. Pour aller plus loin, on regroupe les classes en **paquets**[10]. Par définition, **un paquet est donc un ensemble de classes**. Les paquets sont utilisés de façon systématique par tous les programmes `Java` comportant plus d'une classe.

La construction d'un paquet est une opération assez technique que nous ne détaillerons pas ici. Le lecteur intéressé trouvera des détails dans [7], [4] et [6] (le chapitre 7 de ce dernier ouvrage est entièrement consacré aux paquets).

Il est cependant utile de savoir utiliser des paquets créés par d'autres programmeurs, par exemple ceux qui sont directement inclus dans `Java`, d'autant plus que c'est relativement simple.

3.7.2 Nom de paquet

Chaque paquet est identifié par un nom. Le nom d'un paquet est constitué d'une suite d'identificateurs `Java`, séparés par le symbole point (`.`). Le paquet standard de `Java` porte par exemple le nom `java.lang`. Il existe des conventions assez complexes pour le nom de paquet. Pour simplifier, on peut dire que les identificateurs employés dans les noms de paquet respectent les mêmes conventions que les noms de variables.

3.7.3 Nom complet d'une classe

Principe

Pour l'instant, nous avons toujours utilisé directement une classe sans jamais se poser de questions au sujet de son nom. Nous avons supposé qu'une classe se manipulait en utilisant directement son identificateur (comme indiqué par exemple à la section 3.1.3).

En général, ce n'est pas le cas. En effet, chaque classe appartient à un paquet et on doit normalement utiliser le nom complet de la casse pour pouvoir la manipuler. Ce nom complet s'obtient en donnant le nom du paquet, suivi d'un point, suivi du nom de la classe. Considérons par exemple la classe `Math` (cf la section 3.4.2). Cette classe appartient au paquet standard `java.lang`.

[10] Le terme anglais est *package* qui peut se traduire par paquetage ou par paquet.

Son nom complet est donc `java.lang.Math`. Si on souhaite utiliser la méthode `sin` de la classe `Math`, on doit donc en théorie écrire `java.lang.Math.sin`, ce qui n'est pas toujours très pratique.

Les abréviations automatiques

Dans tout programme `Java`, on peut utiliser directement toutes les classes du paquet standard `java.lang` sans avoir à passer par leur nom complet. L'identificateur de la classe est suffisant.

3.7.4 L'importation

Seules les abréviations automatiques présentées dans la section précédente sont directement utilisables. Cela signifie que pour utiliser une classe d'un paquet différent du paquet standard, il faut normalement utiliser son nom complet. Considérons par exemple le paquet `dauphine.util`. Pour utiliser la classe `Console` de ce paquet, il faut normalement écrire `dauphine.util.Console`. Voici un exemple d'utilisation du nom complet :

Exemple 3.15 :

NomComplet

```
1  public class NomComplet {
2    public static void main(String[] args) {
3      dauphine.util.Console.start();
4      int i=dauphine.util.Console.readInt();
5      System.out.println(2*i);
6    }
7  }
```

Les noms complets sont relativement lourds dans une utilisation intensive. Fort heureusement, `Java` permet l'utilisation d'abréviations, par un mécanisme appelé **importation**. Comparons l'exemple précédent avec ce que nous avons appris à la section 3.5.4 concernant les saisies : on remarque qu'il manque la première ligne `import dauphine.util.* ;` dans laquelle on retrouve le nom du paquet `dauphine.util`. Le but de cette première ligne est d'autoriser l'utilisation directe de toutes les classes contenues dans le paquet `dauphine.util`, sans avoir besoin de passer par leur nom complet.

Considérons de façon plus générale la ligne suivante :

`import` *nom de paquet*`.*;`

Elle autorise le fichier qui la contient à utiliser directement toutes les classes contenues dans le paquet nommé *nom de paquet*, sans avoir à passer par leur nom complet. Un fichier donné peut contenir autant de ligne `import` que souhaité. On peut d'ailleurs utiliser une forme plus restrictive :

`import` *nom de paquet*`.`*nom de classe*`;`

Cette ligne autorise le fichier qui la contient à utiliser directement la classe de nom complet *nom de paquet*`.`*nom de classe*. Cette classe pourra donc être manipulée en écrivant directement *nom de classe*. Comme nous n'utilisons que la classe `Console` du paquet `dauphine.util`, le programme de l'exemple 3.15 peut être réécrit de la façon suivante :

Exemple 3.16 :

Importation

```
1  import dauphine.util.Console;
2  public class Importation {
3    public static void main(String[] args) {
4      Console.start();
```

```
5       int i=Console.readInt();
6       System.out.println(2*i);
7     }
8   }
```

Le fait de pouvoir importer une seule classe est assez utile quand deux paquets définissent des classes de même nom.

3.8 Conseils d'apprentissage

Ce chapitre comporte à la fois une description des mécanismes d'utilisation des méthodes et une longue liste de méthodes (et de constantes) utiles. Les remarques suivantes tentent de résumer les points importants pour permettre au lecteur d'organiser son apprentissage :

- Il faut tout d'abord bien comprendre que l'**appel de méthode** n'a rien de mystérieux : il demande simplement au processeur d'exécuter les instructions qui constituent la méthodes.
- Il est très important de bien comprendre les mécanismes de **typage** et d'**évaluation** d'un appel de méthode. Ce n'est pas très difficile, car il s'agit essentiellement des mêmes règles que pour le typage et l'évaluation des expressions.
- Certaines méthodes doivent impérativement être connues et maîtrisées :
  - sans **affichage**, il est impossible d'écrire un programme utile. Il faut donc connaître les méthodes `print` et `println` ;
  - sans **saisie**, il est difficile d'écrire un programme utile. Il faut donc connaître les méthodes de la classe `Console` ;
  - il est très utile de connaître les méthodes mathématiques élémentaires de la classe `Math` (calcul des fonctions trigonométriques, de la valeur absolue, etc.).
- L'affichage de texte, ou plus précisément l'affichage combiné de texte et du contenu de variables est assez délicat. Il faut être très attentif à l'interprétation de l'opérateur de concaténation.
- Le principe des **constantes** est très simple et doit donc être retenu. Les constantes sont assez utiles, car elles permettent d'écrire un programme de façon plus lisible : quand on utilise `Short.MAX_VALUE`, tout le monde comprend qu'il s'agit de la plus grande valeur possible pour le type `short`. Quand on écrit `32767`, c'est bien moins clair.
- Les éléments contenus dans ce chapitre et les précédents permettent de commencer à lire la documentation `Java` [12]. De nombreuses classes sont disponibles (organisées en paquets) et permettent de réaliser des traitements sans avoir à les programmer. Il est donc utile de lire cette documentation quand on doit résoudre un problème. La classe `Character` par exemple définit de nombreuses méthodes de manipulation des `char`s.

CHAPITRE 4

Structures de sélection

Sommaire

Introduction

Les programmes que nous pouvons maintenant écrire restent assez limités. Leur limitation est principalement due au comportement de la tête de lecture qui parcourt le programme. Nous avons dit qu'elle lit le programme instruction après instruction et que le processeur exécute ainsi les instructions les unes à la suite des autres.

Pourquoi un tel modèle est-il extrêmement limité ? Simplement car lorsqu'on écrit le moindre programme utile, les instructions qui sont exécutées dépendent des informations fournies par l'utilisateur. Pour bien le comprendre, étudions un exemple très simple, celui de la résolution d'une équation du premier degré.

Nous souhaitons donc trouver la ou les solutions d'une équation de la forme $ax + b = 0$. Pour ce faire, écrivons les lignes suivantes qui constituent le début d'un programme :

equation-saisie

```
1 Console.start();
2 double a,b;
3 // saisie des coefficients
4 System.out.print("Coefficient de X : ");
5 a=Console.readDouble();
6 System.out.print("Coefficient constant : ");
7 b=Console.readDouble();
```

Et maintenant, que faire ? Tout dépend des valeurs numériques de `a` et `b`. En effet, si `a` est nul, on ne peut pas calculer[1] `b/a`. Il nous faut donc un moyen de choisir quelles instructions seront exécutées en fonction, par exemple, de la valeur d'une variable ou du résultat d'un calcul. Ce moyen s'appelle une **instruction de sélection** dont nous allons étudier les variantes dans de chapitre.

[1] On peut faire le calcul, mais le résultat n'est pas directement utilisable.

Nous commencerons par l'instruction `if else` qui est la plus utile. Cette instruction permet au processeur de choisir entre deux alternatives. Pour tirer pleinement parti du `if else`, nous introduirons la notion de **bloc**, une construction syntaxique qui permet de regrouper des instructions et donc de manipuler des morceaux de programme. Nous étudierons ensuite l'instruction `if`, une variante du `if else` qui permet une exécution conditionnelle d'une instruction : en fonction d'une condition, le processeur décide ou non d'exécuter une certaine instruction.

Nous aborderons ensuite des outils très généraux, qui s'utilisent avec tous les langages de programmation. Nous parlerons tout d'abord d'**algorithme**. Un algorithme est la version informatique de la recette de cuisine : une suite de tâches élémentaires à mettre en œuvre pour obtenir un certain résultat. Grâce à un algorithme, on peut décrire une méthode de résolution d'un problème qui pourra ensuite être programmée en `Java` comme dans un autre langage. Nous verrons aussi la notion d'**organigramme**, qui est une technique de représentation graphique de la structure logique d'un programme. Son principal intérêt est de faciliter la compréhension des programmes complexes.

Nous terminerons ce chapitre par la présentation de l'instruction `switch`. Celle-ci permet une programmation simplifiée d'une sélection entre plus de deux alternatives : le processeur peut choisir entre un nombre arbitraire de morceaux de programmes en fonction d'une valeur entière.

4.1 La sélection

4.1.1 Sélection entre deux instructions

Principe

On dispose dans tous les langages de programmation d'une **instruction composée** qui permet de réaliser une sélection entre deux instructions différentes. Une instruction composée est en fait une instruction obtenue en plaçant des mots-clés de construction devant une (ou plusieurs) autre(s) instruction(s), les sous-instructions. Nous découvrirons progressivement de telles instructions.

Le principe de la sélection entre deux instructions est simple : le processeur évalue une expression booléenne. Si le résultat est `true` il exécute la première sous-instruction puis passe à la suite du programme. Si le résultat est `false` il exécute la seconde sous-instruction puis passe à la suite du programme. L'instruction de sélection de `Java` est l'instruction `if else`. La forme générale de l'utilisation du `if else` est la suivante :

```
if (expression de type boolean)
  instruction 1;
else
  instruction 2;
```

Le processeur exécute cette instruction de la façon suivante :

1. il évalue l'expression booléenne ;
2. si l'expression vaut `true`, il exécute l'instruction 1 et avance directement la tête de lecture à l'instruction qui suit le `if else` ;
3. sinon, il ne tient pas compte de l'instruction 1, exécute directement l'instruction 2 et avance ensuite la tête de lecture à l'instruction qui suit le `if else`.

Voici un exemple simple d'utilisation de la sélection :

Exemple 4.1 :

On considère le programme :

IfElseSimple

```
1  public class IfElseSimple {
2    public static void main(String[] args) {
3      double x=Math.random();
4      if(x<0.5)
5        System.out.println(x+" est inférieur strictement à 0.5");
6      else
7        System.out.println(x+" est supérieur ou égal à 0.5");
8      System.out.println("Fin du programme");
9    }
10 }
```

L'affichage obtenu est aléatoire, car il dépend de la valeur que contient `x`. En effet, supposons que `x` contienne une valeur strictement inférieure à 0.5. Dans ce cas, l'expression `x<0.5` a pour valeur `true`. D'après la définition de l'instruction `if else`, le processeur exécute alors l'instruction de la ligne 5, puis passe directement à la ligne 8. On obtient alors un affichage de la forme suivante :

AFFICHAGE

```
0.4642036317612983 est inférieur strictement à 0.5
Fin du programme
```

Si, au contraire, `x` contient une valeur supérieure ou égale à 0.5, l'expression `x<0.5` a pour valeur `false`. Dans ce cas, le processeur passe directement à la ligne 7, exécute l'instruction considérée, puis passe à la ligne 8. On obtient alors un affichage de la forme suivante :

AFFICHAGE

```
0.797173875462679 est supérieur ou égal à 0.5
Fin du programme
```

L'affichage de "Fin du programme" n'a ici qu'un seul intérêt : bien montrer qu'une fois l'alternative choisie, le processeur reprend une exécution normale.

La dénomination *instruction de sélection* est parfaitement adaptée, puisqu'il s'agit ici de choisir entre deux instructions à exécuter.

Le point-virgule

Quand on utilise une instruction de sélection, il faut être très attentif à l'emploi du point-virgule, comme le montre l'exemple suivant :

Exemple 4.2 :

Le programme suivant semble parfaitement correct :

MauvaisPointVirgule

```
1  public class MauvaisPointVirgule {
2    public static void main(String[] args) {
3      double x=Math.random();
4      if(x>0.5);
5        System.out.println("Supérieur strictement à 0.5");
6      else
7        System.out.println("Inférieur à 0.5");
```

```
8     }
9   }
```

Pourtant le compilateur le refuse en donnant le message suivant :

ERREUR DE COMPILATION

```
MauvaisPointVirgule.java:6: 'else' without 'if'
    else
    ^
1 error
```

Toute l'erreur provient du point-virgule qui termine la ligne 4. Pour le compilateur, ce symbole représente **ici** une instruction vide. De ce fait, l'affichage de la ligne 5 ne fait pas partie de l'instruction composée `if else`. Le compilateur croit avoir à faire à une instruction composée `if` sans `else` (voir la section 4.1.3). Quand il rencontre le `else` sur la ligne 6, le compilateur ne comprend plus vraiment le programme, d'où le message d'erreur.

Les `if else` emboîtés

Nous avons vu que l'instruction composée `if else` forme une seule instruction. De ce fait, il est possible d'utiliser un `if else` comme composant d'un autre `if else`, comme l'illustre l'exemple suivant :

Exemple 4.3 :

On considère le programme suivant :

IfEmboites

```
1  import dauphine.util.*;
2  public class IfEmboites {
3    public static void main(String[] args) {
4      Console.start();
5      int i=Console.readInt();
6      if(i<0)
7        System.out.println("valeur strictement négative");
8      else
9        if(i>0)
10         System.out.println("valeur strictement positive");
11       else
12         System.out.println("valeur nulle");
13     System.out.println("Terminé");
14   }
15 }
```

Le `if else` qui commence à la ligne ligne 9 et termine à la ligne 12 incluse, constitue une seule instruction. De ce fait, il peut être utilisé comme sous instruction du premier `if else`. L'interprétation du programme est relativement simple. Si l'utilisateur saisit une valeur strictement négative, l'expression booléenne de la ligne 6 prend la valeur `false`. Le processeur exécute donc la ligne 7, puis passe directement à la ligne 13.

Si la valeur saisie est positive ou nulle, l'expression de la ligne 6 vaut `false`. Le processeur passe donc à la ligne 9, qu'il exécute comme un `if else` classique. Si la valeur saisie est nulle, l'expression de la ligne 9 vaut `false`. Le processeur exécute donc la ligne 12, puis passe à la ligne 13.

4.1.2 Les blocs

Définition

Le `if else` reste en l'état assez limité, car le processeur ne peut choisir ici qu'entre deux instructions. Or, dans la pratique, il faut pouvoir choisir entre deux *groupes* d'instructions, ce qui passe par la notion de **bloc** :

Définition 4.1 *Un **bloc** est une suite d'instructions encadrées par une paire d'accolades. La forme générale est donc :*

```
{
  instruction 1;
  instruction 2;
  ...
  instruction n;
}
```

En `Java`*, on peut toujours remplacer une instruction seule par un bloc.*

Application

Toute l'astuce vient du fait qu'une instruction peut toujours être remplacée par un bloc. En écrivant une sélection entre deux blocs, on réalise de fait une sélection entre deux morceaux de programme, comme l'illustre l'exemple suivant :

Exemple 4.4 :

On considère le programme suivant :

IfElseAvecBlocs

```
1  public class IfElseAvecBlocs {
2    public static void main(String[] args) {
3      double x=Math.random();
4      double y;
5      if(x<0.5) {
6        y=-Math.random();
7        x=x+0.5;
8      } else {
9        y=Math.random();
10       x=x-0.5;
11     }
12     System.out.println(x+" "+y);
13   }
14 }
```

Suivant la valeur de `x`, on obtient des résultats assez différents. En effet, quand `x` contient initialement une valeur strictement inférieure à 0.5, la valeur de `y` est alors choisie aléatoirement dans l'intervalle $[-1, 0]$. De plus, on ajoute 0.5 à la valeur de `x`. De ce fait, la valeur de `x` est maintenant élément de l'intervalle $[0.5, 1[$. On obtient par exemple l'affichage suivant :

AFFICHAGE

```
0.9246319856132303 -0.7810134323681446
```

Si au contraire `x` contient initialement une valeur supérieure ou égale à 0.5, on choisit `y` aléatoirement dans l'intervalle $[0, 1]$ et on retranche 0.5 à `x`, dont la valeur est maintenant élément de l'intervalle $[0, 0.5]$. On obtient par exemple l'affichage suivant :

AFFICHAGE

```
0.31202449436476865 0.1601916806061321
```

Blocs emboîtés

Comme on peut remplacer n'importe quelle instruction par un bloc, les blocs peuvent être **emboîtés**. On peut donc avoir un bloc à l'intérieur d'un bloc, comme dans l'exemple suivant :

Exemple 4.5 :

```
{
  int i=2;
  i = i+2;
  {
    int j=i;
    j = j+i;
  }
}
```

Le bloc le plus interne de cet exemple est dit **emboîté** dans le bloc le plus externe. Tout ce passe comme si un bloc était une "boîte". Le gros bloc contient donc une petite "boîte" qui forme un sous-bloc. Il faut bien sûr qu'un bloc soit fermé pour pouvoir être inclus dans un autre : on ne peut pas avoir de blocs se chevauchant.

REMARQUE

Nous remarquons que le sous-bloc n'est pas terminé par un point virgule alors que dans notre présentation, nous indiquons toujours un point virgule après une instruction. En fait, le point virgule fait partie de l'instruction simple. Par contre, le bloc est clairement délimité par les accolades et la point virgule n'en fait pas partie. De ce fait, quand on écrit `{...};`, le processeur interprète le point virgule comme une instruction vide qui suit le bloc. Le point virgule à la fin d'un bloc est donc totalement inutile, et même parfois néfaste, comme l'illustre l'exemple qui suit cette remarque.

REMARQUE

Reprenons l'exemple 4.2, mais en utilisant des blocs. On propose le programme suivant :

MauvaisPointVirguleBloc

```
1  public class MauvaisPointVirguleBloc {
2    public static void main(String[] args) {
3      double x=Math.random();
4      if(x>0.5) {
5        System.out.println("Supérieur strictement à 0.5");
6      };
7      else {
8        System.out.println("Inférieur à 0.5");
9      }
10   }
11 }
```

Ce programme ne compile pas et le compilateur donne le message d'erreur suivant :

Erreur de compilation

```
MauvaisPointVirguleBloc.java:7: 'else' without 'if'
    else {
    ^
1 error
```

L'erreur est identique à celle obtenue dans l'exemple 4.2. En effet, comme le compilateur considère le point virgule comme une instruction vide, la ligne 6 contient à la fois la fin d'un bloc et une instruction. De ce fait, le compilateur pense être en présence d'un `if` seul (voir la section 4.1.3).

Conventions pour les blocs

Les exemples de bloc donnés dans cette section respectent des conventions de présentation qui rendent les programmes plus facile à lire :

1. l'accolade ouvrante qui débute un bloc est placée à la fin de la ligne qui doit ouvrir le bloc ;
2. les instructions qui forment le bloc sont décalées d'un cran ;
3. l'accolade fermante qui termine un bloc est en générale seule sur une ligne. Elle toujours est alignée avec le début de la ligne qui contient l'accolade ouvrante qui lui correspond ;
4. dans le cas du `else`, on place traditionnellement l'accolade fermant le premier bloc, le `else` et l'accolade ouvrant le second bloc sur une même ligne.

Il est vivement conseillé de toujours utiliser des blocs, même quand il est théoriquement possible de s'en passer. Dans le cas du `if else` par exemple, il est déconseillé d'utiliser la forme simple donnée dans la section précédente.

Un exemple plus réaliste

Nous pouvons maintenant résoudre le problème évoqué en introduction, à savoir calculer la ou les solutions de l'équation $ax + b = 0$. Pour résoudre ce problème, il faut traiter séparément les différents cas possibles :

- a n'est pas nul :
  Dans ce cas, l'équation possède une unique solution. Le morceau de programme suivant permet de calculer cette solution (il est censé se placer après la partie de programme donnée en introduction) :

equation-solution

```
1 double solution;
2 solution=-b/a;
3 System.out.println("Une solution unique :");
4 System.out.print("solution :"+solution);
```

- a est nul :
  Il faut alors distinguer deux cas :
  - b n'est pas nul : l'équation ne possède pas de solution. Le morceau de programme suivant affiche alors le résultat :
    `System.out.println("L'équation ne possède pas de solution");`
  - b est nul : l'équation possède une infinité de solutions. Le morceau de programme suivant affiche alors le résultat :
    `System.out.println("Tout réel est solution de l'équation");`

Grâce à deux `if else` emboîtés, il est facile de traiter les trois cas. On obtient le programme suivant :

Resolution

```
1  import dauphine.util.*;
2  public class Resolution {
3    public static void main(String[] args) {
4      Console.start();
5      double a,b;
6      // saisie des coefficients
7      System.out.print("Coefficient de X : ");
8      a=Console.readDouble();
9      System.out.print("Coefficient constant : ");
10     b=Console.readDouble();
11     // a est il nul ?
12     if (a!=0) {
13       // une solution unique
14       double solution;
15       solution=-b/a;
16       System.out.println("Une solution unique :");
17       System.out.println("solution :"+solution);
18     } else {
19       if (b!=0) {
20         // pas de solution
21         System.out.println("L'équation ne possède pas de solution");
22       } else {
23         // une infinité de solutions
24         System.out.println("Tout réel est solution de l'équation");
25       }
26     }
27   }
28 }
```

4.1.3 Exécution conditionnelle d'une instruction

On dispose dans tous les langages de programmation d'une **instruction composée** qui permet d'exécuter une instruction si une condition est remplie. En `Java`, on utilise l'instruction composée `if`, sous la forme suivante :

```
if (expression de type boolean)
  instruction;
```

Le processeur exécute cette instruction de la façon suivante :

1. il évalue l'expression booléenne ;
2. si l'expression vaut `true`, il exécute l'instruction qui constitue la deuxième partie du `if`, puis il passe à l'instruction qui suit le `if` complet ;
3. sinon, il passe directement à l'instruction qui suit le `if` sans exécuter la deuxième partie du `if`.

Il s'agit donc tout simplement d'un `if else` auquel on retire le `else`. Il n'y donc plus de sélection entre deux alternatives, mais bien une *exécution conditionnelle*, car une partie du programme est exécutée seulement si une condition est remplie.

Exemple 4.6 :

On souhaite choisir aléatoirement un réel dans l'intervalle $]0,1]$. On sait que la méthode `random` de la classe `Math` renvoie un réel dans l'intervalle $[0,1]$. Il nous faut donc "supprimer" le zéro. Le programme suivant utilise une exécution conditionnelle pour atteindre ce but :

NoZero

```
1  public class NoZero {
2    public static void main(String[] args) {
3      double x=Math.random();
4      if(x==0)
5        x=1;
6      System.out.println(x);
7    }
8  }
```

Dans le cas (exceptionnel, mais possible) où `x` contient la valeur 0, le `if` la remplace par 1. On obtient bien ainsi un réel aléatoire dans l'intervalle $]0,1]$.

Il est bien sûr possible de remplacer l'instruction qui constitue la deuxième partie du `if` par un bloc d'instructions, exactement comme dans un `if else`. Voici un exemple d'utilisation d'un bloc :

Exemple 4.7 :

On souhaite maintenant choisir aléatoirement un réel dans l'intervalle $[0,1]$, mais on veut réduire la probabilité d'obtenir 0 ou 1. Pour ce faire, on va recommencer le choix aléatoire si on obtient 0 ou 1. Si on obtient encore 0 ou 1, on recommencera de nouveau :

AlmostNoZeroAndOne

```
1  public class AlmostNoZeroAndOne {
2    public static void main(String[] args) {
3      double x=Math.random();
4      if(x==0 || x==1) {
5        x=Math.random();
6        if(x==0 || x==1) {
7          x=Math.random();
8        }
9      }
10     System.out.println(x);
11   }
12 }
```

Le bloc qui comporte une seule instruction (celle de la ligne 7) pourrait bien sûr être remplacé par cette instruction, mais l'utilisation d'un bloc rend la lecture plus facile (voir les conventions de présentation, dans la section 4.1.2). Notons que le programme présenté n'est pas très pertinent : il existe de meilleurs techniques pour effectuer la même chose.

4.2 Subtilités dans l'utilisation du `if else` et du `if`

Les instructions composées `if else` et `if` sont relativement simples à utiliser, en grande partie parce que leur interprétation par le processeur (c'est-à-dire leur *sémantique*) est élémentaire. Dans la pratique, ces instructions interagissent avec d'autres constructions de `Java`, provoquant parfois des comportements inattendus. Le but de cette section est d'entrer dans les détails techniques afin d'illustrer les problèmes qu'on peut parfois rencontrer en utilisant un `if else`.

4.2.1 Emploi des booléens

Il est fréquent de voir dans des programmes une utilisation des variables de type `boolean` (c'est-à-dire les valeurs de vérité) qui traduit une incompréhension profonde de ces variables.

Comme nous l'avons dit dans tout le début de ce chapitre une sélection ou une exécution conditionnelle se base sur une expression de type `boolean` pour choisir le morceau de programme qu'elle doit exécuter. Supposons que pour des raisons diverses, la valeur de vérité en question soit contenue dans une variable. Voici un exemple de programme qui utilise une telle variable de façon assez lourde :

BooleanLourd

```
1  import dauphine.util.*;
2  public class BooleanLourd {
3    public static void main(String[] args) {
4      Console.start();
5      int x;
6      x=Console.readInt();
7      boolean t=x>0;
8      if (t==true) {
9        System.out.println("x est positif");
10     } else {
11       System.out.println("x est négatif ou nul");
12     }
13   }
14 }
15
```

Le problème vient de l'expression `t==true`. Bien entendu, cette expression est correcte et le programme proposé fait bien ce qu'on attend de lui. Cependant, l'expression `t==true` effectue un calcul. En effet, elle va comparer le contenu de la variable `t` à la valeur booléenne `true`. Si `t` contient `true`, alors la valeur de l'expression `t==true` sera `true`. Dans le cas contraire, ce sera `false`. De ce fait, écrire `t==true` permet de faire un calcul dont le résultat est le contenu de `t` ! Ceci est particulièrement inutile (et même un peu stupide car on fait faire au processeur un calcul superflu).

En fait, ce genre d'utilisation des booléens traduit une incompréhension de la notion d'expression booléenne. En effet, le `if` doit être suivi d'une **expression booléenne** entre parenthèses et non pas d'une comparaison. De ce fait, il est parfaitement possible de remplacer `t==true` par `t` car la variable `t` seule représente une expression simple dont la valeur est le contenu de la variable en question. La bonne version du programme est simplement :

BooleanOk

```
1  import dauphine.util.*;
2  public class BooleanOk {
3    public static void main(String[] args) {
4      Console.start();
5      int x;
6      x=Console.readInt();
7      boolean t=x>0;
8      if (t) {
9        System.out.println("x est positif");
10     } else {
11       System.out.println("x est négatif ou nul");
```

```
12      }
13    }
14  }
15
```

REMARQUE

L'exemple présenté est assez artificiel, puisqu'on aurait pu se passer de la variable et utiliser directement la comparaison comme expression booléenne du `if else`. Il s'agit bien entendu d'illustrer "l'erreur" sur un cas simple.

Insistons sur le fait qu'il ne s'agit pas *stricto sensu* d'une erreur, car le compilateur accepte le programme qui se comporte ensuite comme prévu. Il s'agit plus d'un symptôme d'un incompréhension concernant les `booleans`, problème qui pourra s'aggraver lors de la suite de l'apprentissage.

4.2.2 Portée d'une déclaration

Il est important de noter qu'on peut déclarer des variables **n'importe où** dans un programme[2]. Cette possibilité est particulièrement intéressante car elle permet de déclarer les variables seulement si on en a besoin. Ce procédé est notamment utilisé dans dans le programme proposé à la section 4.1.2 pour résoudre l'équation $ax+b=0$: chaque bloc résout une partie du problème et n'a aucune raison d'utiliser les mêmes variables que les autres.

Cette technique permet aussi de rendre le programme plus clair dès que celui-ci dépasse une certaine taille. Si on devait en effet déclarer toutes les variables au même endroit[3], on serait obligé de toujours revenir à cette partie du texte pour vérifier les noms des variables, leur type, etc. De plus, on déclarerait aussi des variables inutiles dans certains cas. Par exemple, la variable `solution`, dans le programme `Resolution` de la section 4.1.2, n'est utile que quand il existe une solution à l'équation. Il n'est bien entendu pas très grave d'avoir des variables inutiles dans un programme, mais un programme minimal dans lequel tout est utile est en général plus facile à comprendre.

Cependant, il ne suffit pas qu'une variable soit déclarée pour qu'une instruction puisse l'utiliser. En effet, chaque déclaration de variable(s) possède une **portée**, c'est-à-dire un ensemble d'instructions qui peuvent utiliser la (les) variable(s) déclarée(s). La règle de portée est très simple :

Définition 4.2 *La **portée** d'une déclaration de variable est constituée par les instructions qui suivent cette déclaration et qui sont dans le même bloc qu'elle. On ne peut utiliser une variable que dans une instruction élément de sa portée.*

Voici tout d'abord un exemple assez artificiel, mais qui illustre les différentes erreurs possibles :

Exemple 4.8 :

On considère le programme suivant :

Portee

```
1  public class Portee {
2    public static void main(String[] args) {
3      int i,j;
4      i=0;
5      if(i>0) {
6        j=2;
```

[2]En général, on conseille vivement de déclarer les variables en début de bloc seulement, afin de retrouver rapidement les variables utiles dans un bloc.

[3]Comme dans certains langages peu évolués, comme le C ou le pascal.

```
7       k=j+3;
8       int k;
9       k=i+j;
10    } else {
11      boolean b;
12      if(k>0) {
13        double x=2.5*i,y=-2.4;
14        x=x+y-3.2/i;
15        b=x>j;
16      } else {
17        b=x<y;
18      }
19    }
20    System.out.println(b);
21    System.out.println(i);
22  }
23 }
```

Le compilateur refuse le programme et affiche de nombreux messages d'erreurs. Voici les raisons de ce refus :

ligne 7 : on utilise ici `k` alors que cette variable n'a pas été déclarée (elle est déclarée à la ligne suivante !) ;

ligne 12 : on utilise `k` dans le second bloc du `if` alors que la déclaration de cette variable est dans le premier bloc et que sa portée se limite donc à ce bloc ;

ligne 17 : même problème que pour l'erreur précédente : les variables `x` et `y` ne peuvent être utilisées que dans le bloc dans lequel elles sont déclarées ;

ligne 20 : idem pour la variable `b`.

REMARQUE

Cet exemple permet de comprendre qu'une variable est bien entendu utilisable dans un sous-bloc du bloc dans lequel elle est déclarée. Un tel sous-bloc est en effet considéré comme une instruction de son sur-bloc et on peut donc utiliser en son sein toute variable déclarée dans le bloc (avant le sous-bloc bien sûr).

Passons maintenant à un exemple plus réaliste :

Exemple 4.9 :

On souhaite déterminer le signe d'un réel donné par l'utilisateur et placer une représentation de ce signe dans une variable entière (on utilise 1 pour un réel positif ou nul, et -1 pour un réel négatif). Croyant bien faire, le programmeur déclare la variable `signe` au dernier moment, ce qui donne le programme suivant :

PasDeVariable

```
1  import dauphine.util.*;
2  public class PasDeVariable {
3    public static void main(String[] args) {
4      Console.start();
5      double x=Console.readDouble();
6      if(x>=0) {
7        int signe=1;
8      } else {
```

```
9       int signe=-1;
10    }
11    System.out.println(signe);
12  }
13 }
```

Ce programme est refusé par le compilateur qui affiche le message suivant :

Erreur de compilation

```
PasDeVariable.java:11: cannot resolve symbol
symbol  : variable signe
location: class PasDeVariable
    System.out.println(signe);
                       ^
1 error
```

On constate que le compilateur ne trouve pas la variable `signe`. En effet, celle-ci est déclarée dans le premier bloc et dans le second. La variable du premier bloc est **différente** de celle du second. Aucune des deux variables n'est accessible en dehors de son bloc et de ce fait, aucune variable `signe` n'est utilisable à ligne 11.

REMARQUE

On pourrait croire que l'utilisation de blocs dans l'exemple précédent est la source des erreurs. En effet, `int signe=1 ;` est une unique instruction et on pourrait croire qu'on peut l'utiliser seule dans un `if else`. Ce n'est pas le cas, pour des raisons techniques qui dépassent le cadre de cet ouvrage. De ce fait, l'exemple proposé n'a rien d'artificiel et correspond (en simplifié) à des situations réelles.

4.2.3 Valeur initiale d'une variable

Comme nous l'avons dit à la section 2.2.5, une variable ne possède pas de valeur initiale. Avant d'utiliser une variable, on doit donc impérativement lui donner une valeur. Or, on peut avoir des surprises en utilisant le `if`, comme l'illustre l'exemple suivant :

Exemple 4.10 :

Reprenons l'exemple 4.9. Pour éviter tout problème, le progammeur déclare maintenant `signe` avant le `if`, mais il remplace le `if else` par deux `if`, ce qui donne :

PasDeValeur

```
1  import dauphine.util.*;
2  public class PasDeValeur {
3    public static void main(String[] args) {
4      Console.start();
5      double x=Console.readDouble();
6      int signe;
7      if(x>=0) {
8        signe=1;
9      }
10     if(x<0) {
11       signe=-1;
12     }
```

```
13      System.out.println(signe);
14    }
15  }
```

Ce programme est refusé par le compilateur qui donne le message suivant :

ERREUR DE COMPILATION

```
PasDeValeur.java:13: variable signe might not have been initialized
    System.out.println(signe);
                       ^
1 error
```

Le message est assez explicite : le compilateur considère que la variable `signe` peut éventuellement ne pas avoir été initialisée et refuse donc son utilisation à la ligne 13. Ce comportement peut sembler surprenant. En effet, pour le *programmeur*, il est clair que soit `x` contient une valeur inférieure ou égale à 0, soit une valeur strictement supérieure à 0. Or, les deux cas sont traités dans le programme, et donc la variable `signe` aura toujours une valeur.

Le problème dans cette explication est qu'elle est basée sur un **raisonnement**, plus précisément sur un résultat mathématique (élémentaire), à savoir le fait que $\neg(x \geq 0) \Leftrightarrow x < 0$. Or, le compilateur n'est pas capable d'effectuer des raisonnements. On peut même prouver qu'il est impossible de concevoir un programme capable de découvrir des théorèmes mathématiques. De ce fait, le compilateur considère les expressions booléennes `x>=0` et `x<0` comme **totalement indépendantes**. Il ne sait donc pas que le processeur exécutera soit l'instruction de la ligne 8, soit celle de la ligne 11. Il ne peut donc pas assurer que la variable `signe` contiendra bien une valeur quand le processeur exécutera la ligne 13.

Bien entendu, le problème peut être simplement résolu dans l'exemple considéré : il suffit d'utiliser un `if else`.

De façon plus général, il faut retenir la règle suivante : une variable est considérée comme initialisée après l'exécution d'une suite d'instructions si et seulement si **toutes les exécutions possibles** de ces instructions conduisent à donner une valeur à la variable. Pour déterminer toutes les exécutions possibles, le compilateur ne tient pas compte des expressions booléennes qui apparaissent dans les `if` et les `if else`. Il procède de la façon suivante :

- Quand il rencontre un `if` seul, le compilateur ne tient pas compte du contenu du `if`, car il est possible que l'instruction qui compose le `if` ne soit pas exécutée. De ce fait, une initialisation réalisée dans un `if` sans `else` n'est pas prise en compte pour déterminer si la variable étudiée est bien initialisée.
- Quand il rencontre un `if else`, le compilateur considère les deux alternatives. Pour qu'une initialisation soit prise en compte, il faut qu'elle apparaisse dans les deux alternatives de la sélection.

Si nous analysons le programme de l'exemple précédent à la lumière de cette règle, le comportement du compilateur devient clair. En effet, les deux initialisations de `signe` (lignes 8 et 11) apparaissent dans des `if` sans `else`. Elles ne sont pas prise en compte, ce qui explique l'erreur repérée par le compilateur.

REMARQUE

Il très important de noter encore une fois qu'il y a une grosse différence entre la partie statique (la compilation) et la partie dynamique (l'exécution) de la "vie" d'un programme. Il est évident qu'à l'exécution, le processeur ne prend en compte que l'alternative correcte quand il exécute un `if else`. De la même façon, l'instruction qui compose un `if` est exécutée si l'expression booléenne

qui la conditionne a pour valeur `true`. C'est seulement à la compilation que certaines parties du programme ne sont pas prises en compte et seulement pour effectuer certaines vérifications, comme l'initialisation que nous venons de voir.

4.2.4 Redéclaration d'une variable

Contraitement à la tradition dans d'autres langages de programmation (comme le C++), la redéclaration d'une variable est impossible en `Java`. Considérons le programme suivant :

Redeclaration

```
1  public class Redeclaration {
2    public static void main(String[] args) {
3      int i=2;
4      i = i+2;
5      if(Math.random()>0.5) {
6        int i=3;
7        int j=i;
8        j = j+i;
9      }
10   }
11 }
```

Ce programme est incorrect. En effet, on tente de redéfinir la variable `i` alors qu'on est dans un sous-bloc du bloc principal dans lequel elle est déjà déclarée. D'ailleurs le compilateur affiche le message suivant :

Erreur de compilation

```
Redeclaration.java:6: i is already defined in main(java.lang.String[])
      int i=3;
          ^
1 error
```

De façon générale, il est impossible de redéfinir une variable si elle existe déjà. Par contre, quand une variable n'est plus accessible (car on est sorti du bloc dans lequel elle était définie) rien n'empêche de la redéfinir (plus précisément de réutiliser l'identificateur correspondant). Le programme suivant est donc correct :

RedeclarationCorrecte

```
1  public class RedeclarationCorrecte {
2    public static void main(String[] args) {
3      int i=2;
4      i = i+2;
5      if(Math.random()>0.5) {
6        int j=i;
7        j = j+i;
8        i = j;
9      } else {
10       double j=i/2.0;
11       i = (int)j;
12     }
13   }
14 }
```

4.3 Un premier algorithme

4.3.1 Qu'est-ce qu'un algorithme ?

La notion d'algorithme est relativement indépendante de celle d'ordinateur mais elle constitue un des éléments fondamentaux de l'informatique. Un algorithme est une sorte de "recette de cuisine" permettant de résoudre *à coup sûr* un problème.

Plus précisément, un algorithme est une description (éventuellement en français) non ambiguë d'une succession de tâches élémentaires à accomplir afin de résoudre un problème. **Un algorithme n'est pas un programme.** Le but d'un algorithme est justement de décrire une méthode permettant de résoudre un problème *indépendamment de tout ordinateur et de tout langage de programmation.*

4.3.2 Exemple d'un algorithme

Pour bien comprendre ce qu'est un algorithme, le plus simple est d'étudier quelques exemples. Nous commençons ici par l'algorithme de résolution d'une équation du premier degré, dont nous avons donné le programme à la section 4.1.2.

Essayons donc de décrire en français le procédé de résolution d'une telle équation. Un algorithme est une *succession d'étapes*. Nous allons donc décrire les différentes étapes qui permettent d'arriver à la solution de l'équation :

1. **Demander à la personne qui propose le problème les données qui définissent celui-ci.**

   Cette phase est universelle. C'est la phase de saisie des données dans un programme informatique. Comme elle est toujours présente, on a pris l'habitude de la décrire d'une façon formalisée : on donne une liste des informations dont on a besoin pour résoudre le problème en donnant un nom à chacune d'elle. Ici on écrira :

   Données :
   – a coefficient de x dans l'équation
   – b coefficient constant dans l'équation
2. **Si a est différent de** 0

   Dans ce cas, la solution est $-\frac{b}{a}$. Il suffit donc de la calculer.
3. **Si a est nul**

   Dans ce cas, on doit distinguer les deux étapes suivantes :

   (a) **Si b est différent de** 0

   Alors n'y a pas de solution.

   (b) **Si b est nul**

   Alors tout réel est solution de l'équation.

Nous remarquons que certaines étapes ne sont effectuées que si une condition est remplie. De plus, certaines étapes contiennent des sous-étapes. De ce fait, un algorithme est très proche d'un programme. Mais il ne contient pas les détails techniques d'un programme. On ne déclare pas de variable, on ne précise pas comment l'affichage et la saisie sont réalisés, on donne les formules mathématiques pour les calculs sans les transformer en expression, on ne fait pas apparaître de type, etc.

Comme nous l'avons dit précédemment, le but d'un algorithme est de résoudre un problème. Or, ceci signifie en général obtenir un *résultat*. De ce fait, l'écrasante majorité des algorithmes que nous écrirons comportera après la liste des données une phrase décrivant le résultat attendu de

l'algorithme. De plus, chaque étape de l'algorithme produisant le résultat (ou un élément de celui-ci) indiquera précisément qu'elle produit un résultat. Avec ces conventions, l'exemple précédent devient :

Données :
- a coefficient de x dans l'équation
- b coefficient constant dans l'équation

Résultat : la solution (si elle existe) de l'équation $ax + b = 0$.

1. **Si** a **est différent de** 0
   Résultat : $-\frac{b}{a}$.
2. **Si** a **est nul**
   (a) **Si** b **est différent de** 0
   Résultat : pas de solution.
   (b) **Si** b **est nul**
   Résultat : tout réel est solution de l'équation.

4.3.3 Présentation graphique d'un algorithme

On peut utiliser un **organigramme** afin de représenter graphiquement un algorithme (ou un programme) en insistant sur sa structure logique. Dans un organigramme chaque étape est représentée par une boîte (qui contient la description de la tâche). Les tâches conditionnelles sont précédées par un losange contenant la condition. Depuis ce losange partent deux branches et sur chacune d'elle est indiqué la valeur correspondante de la condition. Chaque branche est alors reliée à la tâche conditionnelle correspondante.

Considérons l'algorithme très simple suivant :

Données :
- a une valeur numérique.

1. Si a est positif strictement :
   On affiche "valeur positive strictement".
2. si a est négatif ou nul :
   On affiche "valeur négative ou nulle".

La figure 4.1 donne l'organigramme correspondant. Pour que l'organigramme soit bien clair, on dessine toujours un cercle de début et un cercle de fin. On indique aussi des flèches afin de bien montrer dans quel ordre les tâches sont enchaînées.

La figure 4.2 donne l'organigramme de la résolution d'une équation du premier degré $ax+b = 0$.

4.3.4 Exemple complet de résolution d'un problème

Il s'agit de résoudre le problème suivant : *Ecrire un algorithme permettant de trouver la ou les solutions dans* $\mathbb{C}$ *d'une équation de la forme* $ax^2 + bx + c = 0$ *(où* a*,* b *et* c *sont réels). Traduire l'algorithme en organigramme, puis écrire le programme* `Java` *correspondant.*

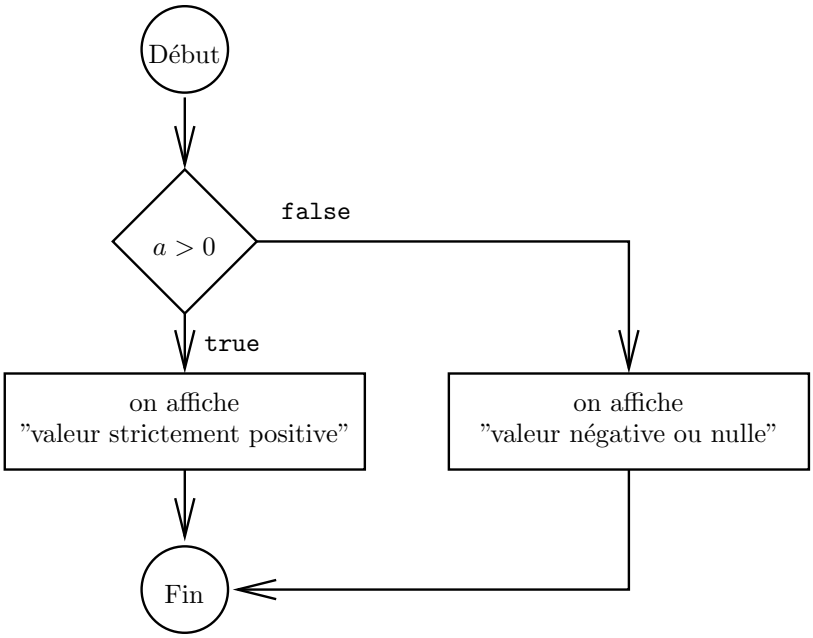

FIG. 4.1 – Organigramme simple

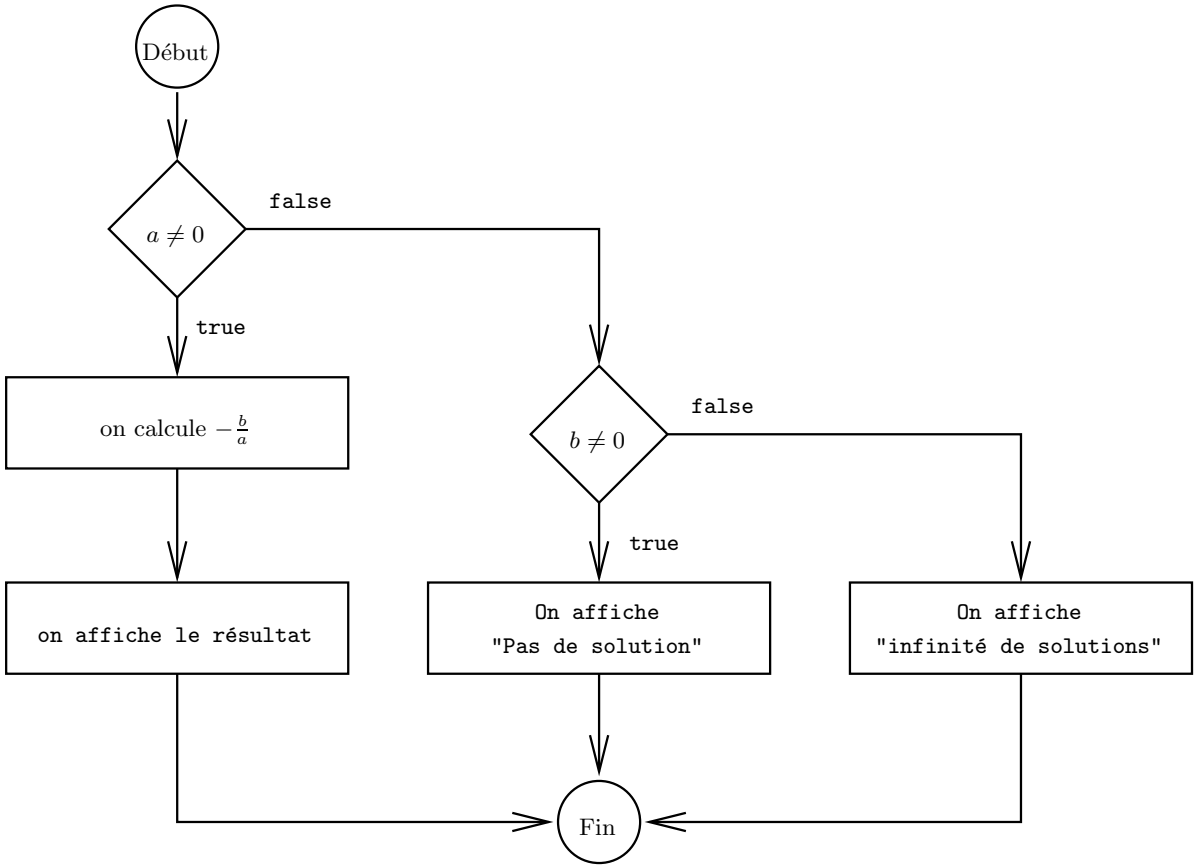

FIG. 4.2 – Organigramme de résolution de $ax + b = 0$

Algorithme

La résolution d'une équation du second degré est relativement simple. On doit tout d'abord vérifier que cette équation est bien du second degré (c'est-à-dire que le coefficient de x^2 n'est pas nul). Ensuite, on calcule le discriminant et suivant son signe, on détermine si l'équation a deux solutions réelles, une solution double ou deux solutions complexes conjuguées. Voilà donc l'algorithme :

Données :
- a coefficient de x^2 dans l'équation
- b coefficient de x dans l'équation
- c coefficient constant dans l'équation

Résultat : les solutions de l'équation $ax^2 + bx + c = 0$.

1. **Si** a **n'est pas nul**
   (a) On calcule le discriminant $\Delta = b^2 - 4ac$.
   (b) **Si** $\Delta > 0$
   On calcule les deux solutions réelles :
      i. On calcule $\sqrt{\Delta}$.
      ii. Résultat : $\frac{-b+\sqrt{\Delta}}{2a}$ et $\frac{-b-\sqrt{\Delta}}{2a}$.
   (c) **Si** $\Delta = 0$
   Résultat : la racine double $-\frac{b}{2a}$.
   (d) **Si** $\Delta < 0$
   On calcule les deux solutions complexes conjuguées :
      i. On calcule $\frac{\sqrt{-\Delta}}{2a}$ et $-\frac{b}{2a}$.
      ii. Résultat : $-\frac{b}{2a} \pm i\frac{\sqrt{-\Delta}}{2a}$.
2. **Si** a **est nul**
   On est ramené au problème déjà traité de la résolution d'une équation du premier degré.

Organigramme

Donnons maintenant un organigramme simplifié de l'algorithme proposé. Pour rendre le dessin plus lisible, on ne recopiera pas la partie concernant la résolution de l'équation du premier degré. La figure 4.3 donne cet organigramme. On remarque l'utilisation de deux cercles de fin pour simplifier le dessin.

Solution complète

Nous pouvons maintenant écrire complètement le programme de résolution d'une équation du second degré. La seule simplification par rapport à un programme vraiment complet est qu'on affiche ici un message pour indiquer que l'équation n'est pas du second degré quand $a = 0$, au lieu de résoudre l'équation du premier degré qui en découle.

Resolution2

```
1 import dauphine.util.*;
2 public class Resolution2 {
3   public static void main(String[] args) {
4     Console.start();
5     double a,b,c,discriminant;
```

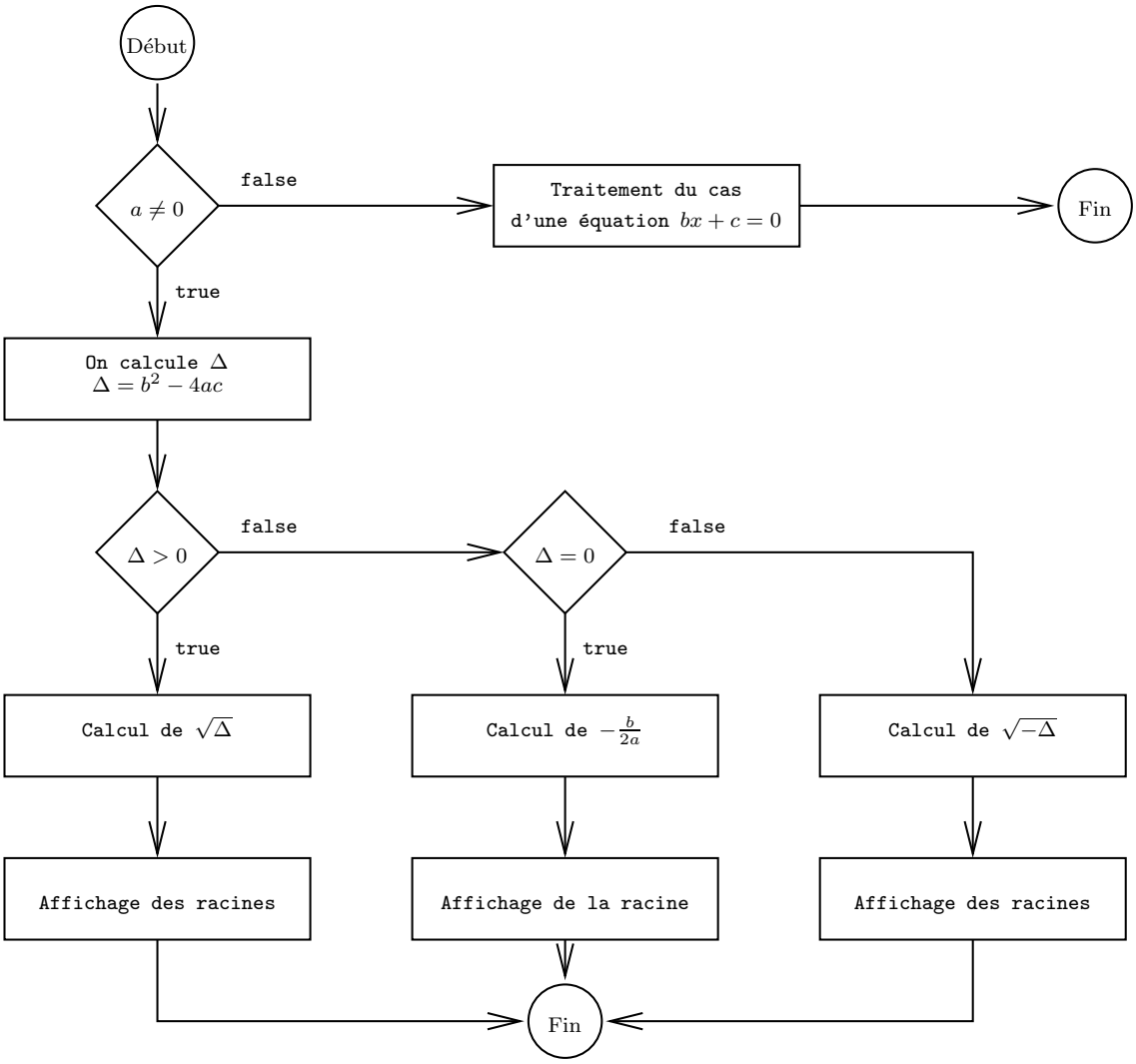

FIG. 4.3 – Organigramme de résolution de $ax^2 + bx + c = 0$

```
6     // saisie des coefficients
7     System.out.print("Coefficient de X^2 : ");
8     a=Console.readDouble();
9     System.out.print("Coefficient de X : ");
10    b=Console.readDouble();
11    System.out.print("Coefficient constant : ");
12    c=Console.readDouble();
13    if (a==0) {
14      System.out.println("L'équation n'est pas du second degré");
15    } else {
16      // calcul du discriminant
17      discriminant=b*b-4*a*c;
18      if (discriminant>0) {
19        // discriminant strictement positif
20        double solution1,solution2,root;
21        root=Math.sqrt(discriminant);
22        solution1=(-b+root)/(2*a);
23        solution2=(-b-root)/(2*a);
24        System.out.println("Deux solutions réelles :");
25        System.out.println("solution 1 :"+solution1);
26        System.out.println("solution 2 :"+solution2);
27      } else {
28        if (discriminant<0) {
29          // discriminant strictement négatif
30          double realPart,imaginaryPart,root;
31          root=Math.sqrt(-discriminant);
32          realPart=-b/(2*a);
33          imaginaryPart=root/(2*a);
34          System.out.println("Deux solutions imaginaires conjuguées :");
35          System.out.println("solution 1 :"+realPart+"+i*"+imaginaryPart);
36          System.out.println("solution 2 :"+realPart+"-i*"+imaginaryPart);
37        } else {
38          //discriminant nul
39          double solution;
40          solution=-b/(2*a);
41          System.out.println("Une solution double réelle :"+solution);
42        }
43      }
44    }
45  }
46 }
```

REMARQUE

Il est clair après cet exemple qu'il y a une assez grosse différence entre l'algorithme (relativement court et simple) et la réalisation pratique. Une des différences importantes réside dans le fait qu'un programme effectue une saisie des paramètres et se contente souvent d'un affichage du résultat. Un algorithme considère au contraire que les paramètres sont des données des problèmes et indique une façon d'obtenir le résultat, sans se servir de celui-ci (et donc sans l'afficher par exemple).

4.4 Sélection entre plusieurs alternatives

4.4.1 Motivations

On est parfois amené à effectuer une sélection entre plus de deux alternatives. L'utilisation de `if else` est alors relativement pénible, comme le montre l'exemple suivant :

Exemple 4.11 :

Le programme suivant est relativement artificiel, mais illustre bien la situation : il s'agit de proposer à l'utilisateur un menu, c'est-à-dire un ensemble de choix possible, puis de réagir au choix effectué.

PlusieursCasIf

```
1  import dauphine.util.*;
2  public class PlusieursCasIf {
3    public static void main(String[] args) {
4      Console.start();
5      System.out.println("Choisir une option : a, b, c ou d");
6      char choix=Console.readChar();
7      if(choix=='a') {
8        System.out.println("Choix A");
9      } else {
10       if(choix=='b') {
11         System.out.println("Choix B");
12       } else {
13         if(choix=='c') {
14           System.out.println("Choix C");
15         } else {
16           if(choix=='d') {
17             System.out.println("Choix D");
18           } else {
19             System.out.println("Choix non reconnu");
20           }
21         }
22       }
23     }
24   }
25 }
```

On constate que le programme est relativement lourd et même difficile à lire. On peut utiliser une "astuce" pour le simplifier. On remarque en effet que chaque `else` (excepté le dernier) contient une seule instruction (un `if else`). On peut donc éviter d'utiliser un bloc et rendre plus compacte la présentation, comme dans cette nouvelle version du programme :

PlusieursCasIfSimple

```
1  import dauphine.util.*;
2  public class PlusieursCasIfSimple {
3    public static void main(String[] args) {
4      Console.start();
5      System.out.println("Choisir une option : a, b, c ou d");
6      char choix=Console.readChar();
7      if(choix=='a') {
```

```
8         System.out.println("Choix A");
9       } else if(choix=='b') {
10        System.out.println("Choix B");
11      } else if(choix=='c') {
12        System.out.println("Choix C");
13      } else if(choix=='d') {
14        System.out.println("Choix D");
15      } else {
16        System.out.println("Choix non reconnu");
17      }
18    }
19  }
```

Le programme n'est pas encore parfait, mais il est beaucoup plus lisible. Il est donc vivement conseillé d'utiliser une telle présentation.

Fort heureusement, `Java` propose une instruction particulière qui permet de simplifier le traitement de sélections entre plus de deux alternatives.

4.4.2 Le `switch`

Un exemple

La forme générale d'un `switch` est relativement complexe, c'est pourquoi nous commençons par un exemple :

Exemple 4.12 :

Reprenons l'exemple 4.11 en utilisant un `switch` :

PlusieursCasSwitch

```
1   import dauphine.util.*;
2   public class PlusieursCasSwitch {
3     public static void main(String[] args) {
4       Console.start();
5       System.out.println("Choisir une option : a, b, c ou d");
6       char choix=Console.readChar();
7       switch(choix) {
8       case 'a':
9         System.out.println("Choix A");
10        break;
11      case 'b':
12        System.out.println("Choix B");
13        break;
14      case 'c':
15        System.out.println("Choix C");
16        break;
17      case 'd':
18        System.out.println("Choix D");
19        break;
20      default:
21        System.out.println("Choix non reconnu");
22        break;
```

```
23      }
24    }
25  }
```

Le nouveau programme fonctionne exactement comme les deux programmes de l'exemple 4.11. On devine que la tête de lecture "saute" automatique à l'instruction qui suit le `case` correspondant à la valeur de la variable `choix`.

Forme générale

Voici maintenant la forme générale de l'instruction `switch` :

```
switch(expression de type byte, short, char ou int) {
case expression constante:
  instruction 1;
  ...
  instruction n;
  break;
...
case expression constante:
  instruction 1;
  ...
  instruction p;
  break;
...
default:
  instruction 1;
  ...
  instruction q;
  break;
}
```

Les règles suivantes doivent être respectées :

- les expressions qui suivent les `case`s sont obligatoirement des expressions **constantes** : elles ne peuvent pas faire apparaître de variables. On les appelle les **étiquettes**[4] ;
- les expressions constantes doivent être d'un type compatible avec celui de l'expression qui suit directement le `switch` ;
- chaque `case` peut contenir autant d'instructions que souhaité ;
- le `default` peut contenir autant d'instructions que souhaité ;
- un `switch` peut contenir autant de `case` que nécessaire, à condition que chaque `case` comporte une étiquette unique ;
- un `switch` peut contenir au maximum un `default`.

Si les règles ne sont pas respectées, le programme est réfusé par le compilateur.

Sémantique

Voici comment le processeur interprète un `switch` :

1. le processeur évalue l'expression qui suit directement le `switch` ;
2. si l'étiquette d'un `case` est égale à la valeur de l'expression :

[4] Les *labels* en anglais.

(a) le processeur déplace la tête de lecture directement à la première instruction du `case` considéré ;

(b) le processeur exécute normalement toutes les instructions du `case` ;

(c) quand le processeur rencontre l'instruction `break`, il termine l'exécution du `switch` et passe donc directement à l'instruction qui suit l'accolade fermante du `switch`.

3. si aucune étiquette n'est égale à la valeur de l'expression :

   (a) si le `switch` comporte un `default`, alors :

      i. le processeur déplace la tête de lecture directement à la première instruction du `default` ;

      ii. le processeur exécute normalement toutes les instructions du `default` ;

      iii. quand le processeur rencontre l'instruction `break`, il termine l'exécution du `switch` et passe donc directement à l'instruction qui suit l'accolade fermante du `switch`.

   (b) sinon, le processeur termine l'exécution du `switch` et passe donc directement à l'instruction qui suit l'accolade fermante du `switch`.

On constate que le `default` peut donc être interprété comme une sorte de `else`, alors que chaque `case` correspond à une sorte de `if`.

4.4.3 L'instruction `break`

Nous avons vu que chaque `case` et que l'éventuel `default` d'un `switch` se terminent par l'instruction `break`. Nous avons indiqué que l'exécution de l'instruction `break` termine le `switch`. Il faut bien comprendre que c'est **exactement** le cas. Si on oublie le `break`, le processeur continue l'exécution en passant au `case` suivant, comme l'illustre le prochain exemple :

Exemple 4.13 :

On considère le programme suivant :

SansBreak

```
1  public class SansBreak {
2    public static void main(String[] args) {
3      int i=(int)(Math.round(3*Math.random()));
4      System.out.println(i);
5      switch(i) {
6      case 0:
7        System.out.println("0");
8        break;
9      case 1:
10       System.out.println("1");
11     case 2:
12       System.out.println("2");
13     default:
14       System.out.println("autre");
15     }
16   }
17 }
```

L'entier contenu dans `i` est déterminé au hasard et est élément de l'intervalle $[0, 3]$. Détaillons le comportement du programme pour les différentes valeurs possibles :

0 dans ce cas, le processeur exécute la ligne 7, puis rencontre un `break` : il passe donc directement à la fin du programme. De ce fait, l'affichage produit est le suivant :

AFFICHAGE
```
0
0
```

1 dans ce cas, le processeur exécute la ligne 10. Comme il ne rencontre par de `break`, il continue en exécutant les lignes suivantes. La ligne 11 ne produit aucun effet, alors que la ligne 12 est exécutée normalement. Comme il ne rencontre toujours pas de `break`, le processeur continue son exécution et finit ainsi par la ligne 14. On obtient donc l'affichage suivant :

AFFICHAGE
```
1
1
2
autre
```

2 dans ce cas, le processeur exécute la ligne 12. Comme il ne rencontre par de `break`, il continue en exécutant les lignes suivantes. La ligne 13 ne produit aucun effet, alors que la ligne 14 est exécutée normalement. L'affichage obtenu est :

AFFICHAGE
```
2
2
autre
```

3 dans le cas, le processeur exécute directement la ligne 14, puis termine normalement le `switch`, ce qui donne l'affichage suivant :

AFFICHAGE
```
3
autre
```

Il est vivement **déconseillé** d'utiliser les possibilités ouvertes par l'absence de `break`, sauf dans un cas particulier que nous allons décrire. Reprenons l'exemple 4.12 qui propose à l'utilisateur un menu à quatre choix, les lettres a, b, c et d. Il peut être utile d'accepter indifféremment une réponse en minuscule ou en majuscule. Or, on ne souhaite pas écrire deux fois le traitement de chaque lettre, une fois pour `'a'` et une fois pour `'A'` par exemple (c'est une perte de temps et une source d'erreur). L'exemple suivant propose une solution basée sur l'absence de `break` :

Exemple 4.14 :

Voici la solution :

PlusieursCasSwitchMajuscule
```
1  import dauphine.util.*;
2  public class PlusieursCasSwitchMajuscule {
3    public static void main(String[] args) {
4      Console.start();
5      System.out.println("Choisir une option : a, b, c ou d");
6      char choix=Console.readChar();
7      switch(choix) {
```

```
8       case 'a': case 'A':
9         System.out.println("Choix A");
10        break;
11      case 'b': case 'B':
12        System.out.println("Choix B");
13        break;
14      case 'c': case 'C':
15        System.out.println("Choix C");
16        break;
17      case 'd': case 'D':
18        System.out.println("Choix D");
19        break;
20      default:
21        System.out.println("Choix non reconnu");
22        break;
23      }
24    }
25  }
```

On a donc indiqué plusieurs `case` à la suite. Pour bien comprendre ce qu'il se passe à l'exécution, il suffit de réécrire le programme. Considérons par exemple les lignes 8 à 10. Elles sont équivalentes aux lignes suivantes :

```
case 'a':
case 'A':
  System.out.println("Choix A");
  break;
```

Le comportement du processeur est alors clair. Quand le caractère saisi est `a`, le processeur exécute le contenu du `case 'a' :` qui est en fait vide. Comme il n'y pas de `break`, le processeur continue l'exécution et passe donc au `case` prévu pour `A`, ce qui est exactement ce qu'on souhaite. Grâce au `break`, l'exécution du `switch` se termine. Si le caractère saisi est `A`, le processeur exécute directement le contenu du `case 'A' :`, ce qui est encore une fois le comportement souhaité.

On doit retenir de cet exemple le point suivant : si on souhaite regrouper plusieurs étiquettes pour leur fournir un traitement commun, il suffit d'indiquer les `case`s correspondant les un à la suite des autres, en profitant de l'absence de `break` pour enchaîner les traitements.

4.4.4 Comparaison entre `if else` et `switch`

Si on compare le programme simplifié de l'exemple 4.11 avec le programme de l'exemple 4.12, on constate que la solution avec `if else` est assez similaire à celle utilisant le `switch`. Il est même clair que la deuxième solution sera plus longue à écrire. On peut donc légitimement s'interroger sur l'intérêt du `switch`. En fait, on peut résumer le `switch` comme un cas très particulier d'enchaînement de `if else`. Un enchaînement de `if else` aura toujours plus de possibilité qu'un `switch`.

C'est justement dans cette limitation que réside l'intérêt du `switch` : le programme de l'exemple 4.12 est plus clair que celui de l'exemple 4.11. En effet, quand on maîtrise le `switch`, on interprète très rapidement le programme utilisant un `swicth` : chaque cas à traiter est clairement séparé des autres (par le `case` et le `break`), l'expression utilisée pour prendre la décision apparaît à un seul endroit (juste après le mot clé `switch`), et le mot clé `default` identifie sans erreur possible le traitement par défaut.

Dans le cas de la série de `if else`, la situation est beaucoup moins claire : il faut en effet lire *chaque* expression booléenne pour comprendre comment est défini chaque cas. Les expressions peuvent être arbitraires et en particulier faire intervenir différentes expressions selon le `if`. On peut donc avoir par exemple la première partie de la série de `if else` qui porte sur une variable et la seconde partie qui porte sur une autre variable, ce qui complique grandement l'interprétation.

De plus, comme la prise de décision est séquentielle (on teste la première expression, puis la seconde, etc. jusqu'à atteindre le cas qui nous intéresse), l'expression booléenne qui précède chaque traitement possible ne suffit pas à définir celui-ci. Considérons par exemple le pseudo-programme suivant (on suppose que la variable `x` de type `double` a été déclarée et initialisée avant) :

```
if(x<2) {
  // traitement 1
}  else if(x>5) {
  // traitement 2
}  else if(x<4) {
  // traitement 3
}  else
 // traitement 4
}
```

Le cas correspondant au premier traitement est très clair, il s'agit de tous les réels strictement inférieurs à 2. Pour le deuxième traitement, tout va bien, il s'agit de tous les réels strictement supérieurs à 5. Pour le troisième traitement, il faut tenir compte des deux autres (en fait du premier), car il ne s'agit des réels strictement inférieurs à 4, mais des réels compris dans l'intervalle $[2, 4[$. Enfin, le dernier traitement (le cas par défaut) correspond à tous les cas non traités, c'est-à-dire ici à l'intervalle $[4, 5]$.

Bien entendu, ce genre de prise de décision complexe ne peut pas être programmée grâce à un `switch`, mais c'est justement cette limitation qui permet une compréhension rapide de celui-ci.

REMARQUE

On comprend pourquoi il est vivement conseillé de toujours utiliser l'instruction `break` : elle simplifie grandement l'interprétation du `switch` et renforce donc son intérêt.

4.5 Conseils d'apprentissage

Nous avons abordé dans ce chapitre nos premières instructions `Java` évoluées. La sémantique des sélections reste relativement simple, mais l'interaction avec les autres instructions `Java` peut poser quelques problèmes. Voici les points importants à retenir :

- Il est **impossible** de réaliser un programme vraiment utile sans utiliser l'instruction `if`, qui sera en générale mise en œuvre sous sa forme complète, le `if else`. Il est donc impératif de connaître parfaitement les modalités de son utilisation et son interprétation par le processeur.
- Pour obtenir des programmes lisibles, il est vivement conseillé de toujours utiliser des **blocs**, qui sont de toute manière indispensable pour écrire un programme qui réalise autre chose qu'une simple démonstration de quelques lignes.
- Un programme `Java` n'est que la traduction dans un langage informatique donné d'un **algorithme**. Comme l'invention d'algorithmes évolués est bien plus difficile que la programmation, il est important de savoir comprendre un algorithme pour le traduire en un programme `Java`.
- L'utilisation **d'organigrammes** est permet de mieux comprendre les programmes ou les algorithmes complexes en faisant apparaître graphiquement leur structure logique.

- Pour éviter de perdre du temps à interpréter les réactions du compilateur, il faut bien comprendre l'interaction du `if else` avec **les déclarations et initialisations de variables**. Quand on a compris que le compilateur (l'ordinateur en général) ne peut pas faire de raisonnement, ses réactions deviennnent assez faciles à analyser et surtout à prévenir : il faut donc être particulièrement attentif quand on déclare ou initialise une variable à l'intérieur d'un `if else`.
- Il est important de retenir que le `if` est basé sur une **expression booléenne générale** et pas seulement sur une comparaison, ce qui évite d'écrire des choses inutiles comme `b==true` à la place de `b`.
- L'instruction `switch` est assez pratique dans le cas d'une sélection entre plus de deux alternatives. Les programmes qui l'utilisent sont souvent plus faciles à lire et à interpréter que ceux qui se basent sur une série de `if else`. Les applications du `switch` sont assez restreintes, mais dans son domaine, cette instruction est la meilleure.

CHAPITRE 5

Structures itératives

Sommaire

Introduction

Les structures de sélection que nous avons introduites dans le chapitre précédent permettent d'écrire des programmes plus intéressants et réalisant des tâches un peu plus réalistes qu'au début de notre apprentissage. Nous allons maintenant aborder un autre aspect très important : les **structures itératives**. Les instructions étudiées vont nous permettre de faire exécuter par le processeur plusieurs fois le même morceau de programme.

Une motivation très simple peut justifier le besoin des structures itératives : le lancement d'un programme `Java` est relativement long, car il faut d'abord lancer la machine virtuelle (voir le chapitre 1). Quand un programme doit interagir avec l'utilisateur, son lancement est encore plus lent, car la `Console` doit d'abord apparaître (voir le chapitre 3). De ce fait, il est naturel de chercher à ne lancer qu'une seule fois un programme et de faire en sorte qu'après avoir terminé sa tâche, celui-ci recommence depuis le début. Cette notion de "retour au début" se traduit informatiquement par une notion de répétition : le processeur exécute plusieurs fois les mêmes instructions, en recommençant à chaque fois au début du groupe considéré.

Dans la pratique, la **répétition** comporte de très nombreuses applications qui dépassent largement le cadre de l'exemple qui vient d'être proposé. Elle permet en particulier d'effectuer des calculs complexes qui sont au centre des applications pratiques de l'informatique : prévisions météorologiques, modélisation des matériaux, etc. En fait, sans répétition, l'ordinateur est quasi inutile.

Les structures qui permettent de répéter des opérations portent le nom de **structures itératives**. Dans la pratique, on les désigne par le nom plus commun de **boucles**. Nous commencerons par étudier les boucles **conditionnelles**, c'est-à-dire des structures permettant de répéter des opérations tant qu'une condition est remplie. Nous verrons comment utiliser ces boucles pour réaliser des calculs complexes qui ne pourraient pas être programmés sans boucle.

Nous étudierons ensuite les boucles "**non-conditionnelles**", des structures qui peuvent répéter des opérations un nombre donné de fois. Nous étudierons des applications naturelles de ces structures, montrant qu'elles cachent en fait une structure conditionnelle, et sont surtout utiles comme instrument de simplification des programmes.

Nous consacrerons la fin du chapitre aux subtilités liées à l'utilisation des boucles, qu'elles soient algorithmiques (comme par exemple l'utilisation correcte des boucles emboîtées ou de l'**interruption de boucle**) ou plus spécifique à `Java` (les problèmes de portées des variables par exemple).

5.1 Boucle conditionnelle post-testée

5.1.1 Exemple introductif

Comme nous l'avons indiqué en introduction, une première motivation des structures itératives est de faire recommencer un programme au début de son traitement, sans pour autant le relancer. L'exemple suivant illustre une solution possible :

Exemple 5.1 :

On considère un programme élémentaire qui calcule la moyenne de deux notes. En ajoutant une nouvelle instruction composée (le `do while`), le programme peut réaliser autant de calcul qu'on le souhaite, sans avoir besoin d'être relancé :

Moyenne

```
1  import dauphine.util.*;
2  public class Moyenne {
3    public static void main(String[] args) {
4      Console.start();
5      do {
6        System.out.print("Première note = ");
7        double note1=Console.readDouble();
8        System.out.print("Seconde note = ");
9        double note2=Console.readDouble();
10       System.out.println("Moyenne = "+( (note1+note2)/2 ));
11       System.out.print("Voulez-vous recommencer ? [o/n] ");
12     } while (Console.readChar()=='o');
13   }
14 }
```

Si on fait abstraction pour l'instant des lignes 5 et 12, ce programme est assez classique. D'ailleurs, quand on l'exécute, il fonctionne tout à fait classiquement jusqu'à la ligne 11 incluse : l'utilisateur saisit deux réels puis le programme affiche la moyenne de ces nombres. Ensuite, il affiche le texte de la ligne 11.

L'utilisateur peut alors saisir un caractère. S'il saisit un caractère différent de la lettre o, le programme s'arrête. Par contre, s'il saisit la lettre o, le programme recommence à la ligne 6. On obtient par exemple l'affichage suivant :

Affichage

```
Première note = 12
Seconde note = 15
Moyenne = 13.5
Voulez-vous recommencez ? [o/n] o
```

```
Première note = 18
Seconde note = 16.5
Moyenne = 17.25
Voulez-vous recommencez ? [o/n] n
```

Le bloc qui commence à la ligne 5 et qui termine à la ligne 12 est donc exécuté une première fois. Puis, si l'expression booléenne qui suit le mot clé `while` vaut `true`, le processeur recommence l'exécution du bloc, etc. *tant que* l'expression vaut `true`.

5.1.2 L'instruction `do while`

Forme générale (syntaxe)

Pour répéter l'exécution d'un morceau de programme, on peut utiliser l'instruction composée `do while`. La forme générale d'utilisation de l'instruction `do while` est la suivante :

```
do
  instruction ;
while (expression de type boolean);
```

Dans la pratique, on conseille vivement de toujours utiliser un bloc, plutôt qu'une seule instruction, ce qui donne la forme suivante :

```
do {
  instruction 1 ;
  ...
  instruction n ;
}  while (expression de type boolean);
```

Sémantique

Quand il rencontre une instruction `do while`, le processeur l'exécute de la façon suivante :

1. il commence par exécuter l'instruction qui suit le `do` (dans le cas d'un bloc, il exécute toutes les instructions du bloc, de façon classique) ;
2. il évalue l'expression qui suit le `while` ;
3. si l'expression vaut `true`, l'exécution recommence en 1 ;
4. sinon, l'exécution de l'instruction `do while` est terminée.

REMARQUE

L'instruction `do while` réalise une *boucle conditionnelle post-testée*. Comme toutes les structures itératives, c'est bien sûr une boucle. Elle est conditionnelle car la répétition n'a lieu que si une condition est remplie. Enfin elle est post-testée car l'évaluation de la condition a lieu *après* l'exécution de l'instruction (ou du bloc) qui se répète.

On comprend maintenant le fonctionnement du programme de l'exemple 5.1. En effet, pour évaluer l'expression de la ligne 12, à savoir `Console.readChar()=='o'`, le processeur doit bien entendu effectuer l'appel de méthode `readChar`. Il doit donc attendre que l'utilisateur saisisse un caractère. Quand cette saisie est effectuée, le caractère obtenu est comparé à la lettre o. Si l'utilisateur a tapé cette lettre, le processeur recommence l'exécution du bloc à la ligne 6, sinon, le programme est terminé.

REMARQUE

L'évaluation de l'expression du `while` est effectuée après chaque exécution de l'instruction (ou du bloc) qui suit le `do`. Dans l'exemple 5.1, cela signifie qu'une saisie est effectuée après chaque exécution du bloc. Le processeur ne se souvient pas de l'évaluation précédente.

Vocabulaire

On utilise un vocabulaire assez spécifique quand on parle de boucle :

corps de la boucle : désigne l'instruction ou le bloc qui est répété par la boucle ;

itération : désigne une exécution du corps de la boucle ;

tour : synonyme d'itération.

Exemples

Commençons par un exemple très simple et très utile dans la pratique :

Exemple 5.2 :

Quand on demande à l'utilisateur de saisir une valeur, il est relativement libre dans sa réponse. Si on lui demande un `int`, il pourra donner n'importe quelle valeur dans l'intervalle acceptable pour les `int`s. Dans certaines situations, ce n'est pas pratique et on utilise alors une boucle pour garantir que la valeur proposée par l'utilisateur satisfait certaines conditions. Si on souhaite par exemple calculer le reste de la division d'un entier par un autre, il faut impérativement que le deuxième entier ne soit pas nul. On propose le programme suivant :

NonNul

```
1  import dauphine.util.*;
2  public class NonNul {
3    public static void main(String[] args) {
4      Console.start();
5      System.out.print("Choisissez un entier : ");
6      int i=Console.readInt();
7      int j;
8      do {
9        System.out.print("Choisissez un entier non nul : ");
10       j=Console.readInt();
11     } while (j==0);
12     System.out.println("Le reste de la division de "+i
13                        +" par "+j+ " est "+(i%j));
14   }
15 }
```

L'instruction `do while` permet de garantir que `j` contient une valeur non nulle. En effet, quand le processeur évalue la condition du `while`, il obtient `true` si la variable `j` contient 0. L'exécution du bloc reprend et l'utilisateur doit saisir une nouvelle valeur pour `j`. Le seul moyen d'exécuter l'instruction de la ligne 12 est que l'instruction `do while` se termine. Or, ceci n'est possible que si l'expression du `while` prend la valeur `false`, ce qui n'est possible que si `j` contient une valeur non nulle. On peut bien sûr adapter cette technique pour n'importe quelle condition.

Voici maintenant un exemple de véritable programme. On est bien sûr encore loin d'un programme utile (ou même intéressant), mais pour une fois, il s'agit d'une véritable application :

Exemple 5.3 :

On souhaite programmer un jeu très simple. On fait choisir à l'ordinateur un entier au hasard dans un certain intervalle, puis l'utilisateur doit deviner ce nombre. L'ordinateur l'aide en lui indiquant si sa proposition est plus petite ou plus grande que le nombre à deviner. Voici un programme qui met en œuvre ce jeu :

Jeu

```
1  import dauphine.util.*;
2  public class Jeu {
3    public static void main(String[] args) {
4      Console.start();
5      do {
6        int max;
7        do {
8          System.out.print("Indiquez la valeur maximale pour le choix ");
9          System.out.print("de l'ordinateur (>0) ");
10         max=Console.readInt();
11       } while(max<=0);
12       int nombre=(int)(max*Math.random());
13       int choix;
14       do {
15         System.out.print("Votre choix ? ");
16         choix=Console.readInt();
17         if(choix>nombre) {
18           System.out.println("Trop grand");
19         }
20         if(choix<nombre) {
21           System.out.println("Trop petit");
22         }
23       } while (choix!=nombre);
24       System.out.println("Bravo, vous avez gagné");
25       System.out.print("Voulez-vous recommencer ? [o/n] ");
26     } while (Console.readChar()=='o');
27   }
28 }
```

Ce programme est relativement long (par rapport à ceux que nous avons vus jusqu'à présent) et demande quelques explications :

- l'affichage en deux instructions (ligne 8 et 9) est utilisé pour faire tenir le texte à afficher dans la présente page. Le résultat est identique à celui obtenu par un affichage en une seule instruction. Il n'y a pas en général de raison d'utiliser cette technique si on ne souhaite pas imprimer le texte du programme ;
- la première instruction `do while`, qui commence à la ligne 5 et termine à la ligne 26, permet de rejouer sans avoir à relancer le programme. On est exactement dans la même situation que dans l'exemple 5.1 ;
- la deuxième instruction `do while` (lignes 7 à 10) correspond à une application de la technique étudiée dans l'exemple 5.2 : on utilise une boucle afin d'être sûr que la variable `max` contiendra une valeur strictement positive dans la suite du programme ;
- la ligne 15 permet de choisir aléatoirement un entier dans l'intervalle $[0, \texttt{max}]$;
- la troisième instruction `do while` (lignes 14 à 23) est la plus importante, puisque c'est elle qui programme le jeu. En effet, l'utilisateur choisit une valeur à la ligne 16. En fonction

de cette valeur, l'ordinateur affiche éventuellement un message d'aide (c'est le but des deux `if`s). Ensuite, la condition de la ligne 23 est évaluée : elle assure que la boucle se termine seulement si l'utilisateur a fait le bon choix. Si le choix est mauvais, le bloc recommence et l'utilisateur doit donc faire un nouveau choix.

5.1.3 Algorithme

Il est important de savoir traduire une description algorithmique correspondant à un traitement répétitif en une boucle `do while`. Voici par exemple l'algorithme du programme de l'exemple 5.3. Comme le programme en question ne calcule rien, l'algorithme proposé n'a pas de résultat :

Donnée :

un entier *max* strictement positif.

1. choisir au hasard un entier *nombre* dans l'intervalle $[0, max]$
2. demander à l'utilisateur un entier *choix*
3. si $choix > nombre$ afficher "trop grand"
4. si $choix < nombre$ afficher "trop petit"
5. si $choix \neq nombre$ recommencer à l'étape 2
6. sinon afficher "gagné"

Bien entendu, l'algorithme proposé ne correspond pas exactement au programme. Il est très simplifié et ne comporte pas la répétition générale (qui permet de rejouer). Ceci est parfaitement normal : au même titre que les saisies, une boucle permettant de reprendre le programme à son début est un "détail" de programmation. Elle est très utile, mais reste un détail. Le point important de l'algorithme est l'étape 5 : elle correspond à la boucle `do while`, puisqu'elle décrit exactement la sémantique de celle-ci. Quand le processeur évalue la condition du `while` et obtient `true`, il bouge sa tête de lecture en arrière, afin de revenir à la première instruction du bloc et recommencer l'exécution de celui-ci.

REMARQUE

Comme nous l'avons déjà indiqué plusieurs fois, les saisies sont des éléments annexes, des détails de programmation, qui ne doivent donc pas apparaître dans l'algorithme. On remarque ici que la saisie utilisant une boucle qui permet d'obtenir une valeur strictement positive pour la variable `max` est tout simplement remplacée par une condition dans les données de l'algorithme.

5.1.4 Organigramme

Pour bien comprendre un programme ou un algorithme utilisant des boucles, il est intéressant d'en donner une représentation graphique par un organigramme. Cela ne pose aucun problème si on se réfère à la sémantique du `do while`. En effet, cette instruction se base simplement sur l'évaluation d'une expression booléenne, qui produit éventuellement un retour en arrière de la tête de lecture du processeur. Il suffira donc de représenter l'expression booléenne par un losange (comme pour les sélections) dont part une flèche qui retourne en arrière dans l'organigramme, comme l'illustre la figure 5.1 qui représente la sémantique du `do while`. La figure 5.2 donne l'organigramme correspondant à l'algorithme de la section précédente (c'est-à-dire à une version simplifié du programme de jeu de l'exemple 5.3). On remarque que la représentation graphique d'une structure itérative forme une boucle au sens classique du terme, c'est-à-dire une suite d'éléments reliés entre eux et dont le dernier est relié au premier. Ceci explique (en partie) pourquoi on désigne les structures itératives sous le nom de boucle.

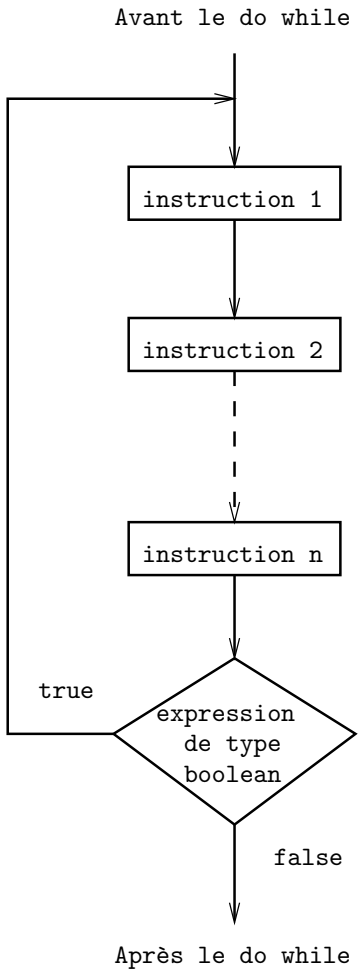

FIG. 5.1 – Organigramme d'un `do while`

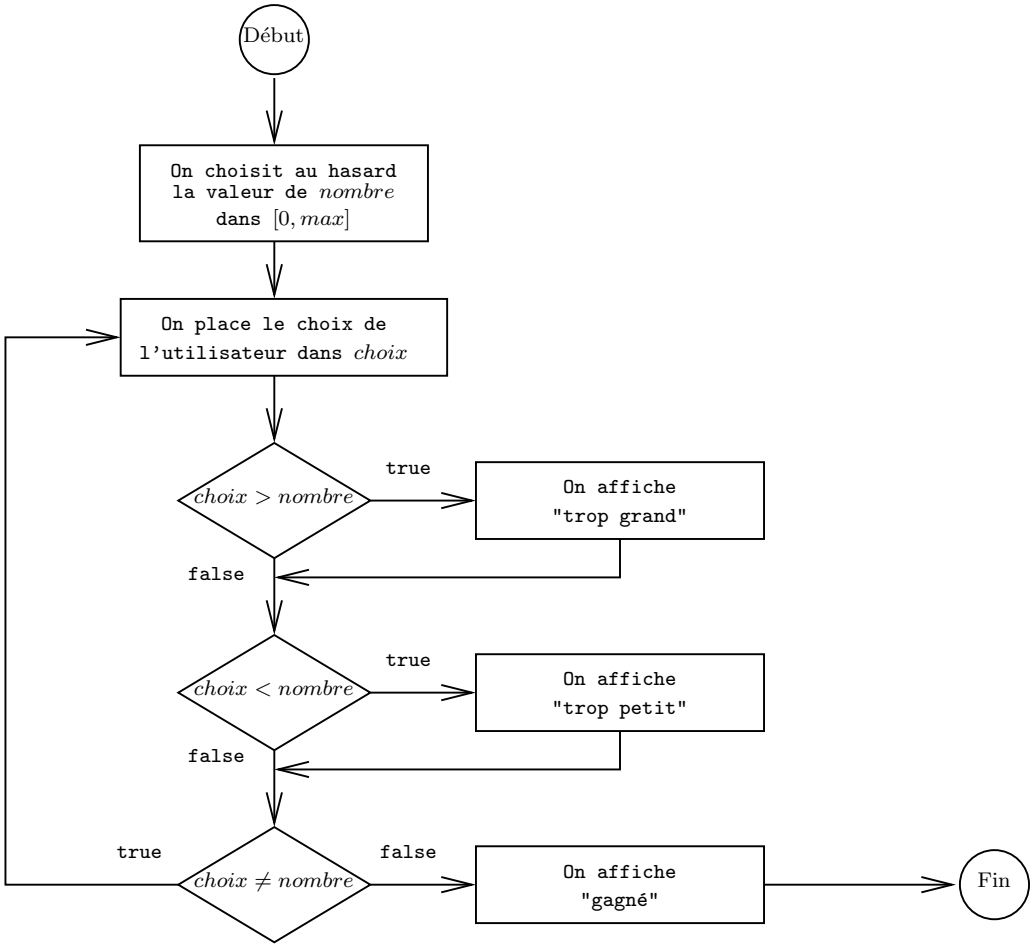

FIG. 5.2 – Organigramme du jeu de découverte d'un nombre

REMARQUE

La représentation graphique par organigramme a le défaut de laisser croire qu'on peut passer d'une partie d'un programme à un autre sans vrai contrôle, par une simple flèche. C'est effectivement le cas dans certains langages (comme le C et le C++), mais tous les informaticiens professionnels s'accordent à dire que l'utilisation de cette possibilité produit *en général*[1] des programmes de mauvaise qualité. Fort heureusement, en `Java`, les mouvements de la tête de lecture ne sont pas arbitraires : en général, on passe d'une instruction à la suivante, sauf dans le cas de constructions clairement identifiables, comme les sélections et les boucles.

5.2 Calculs itératifs

5.2.1 Illustration

Reprenons l'exemple 5.3. Pour l'instant, ce jeu n'est pas très gratifiant car il ne mesure pas la qualité du joueur. Si le nombre à découvrir est compris entre 0 et 100, l'ordinateur ne fait pas de différence entre quelqu'un qui trouve le nombre en 7 essais et quelqu'un qui tente presque toutes les valeurs (en faisant par exemple 50 essais). Il est logique de mesurer la qualité du joueur en comptant le nombre de tentatives effectuées et en le comparant au nombre maximal de tentatives, c'est-à-dire le nombre d'entiers que l'ordinateur peut choisir[2]. Voici une solution qui mesure la qualité du joueur :

Exemple 5.4 :

Dans cette nouvelle version du programme, nous supprimons la boucle de répétition complète, afin de simplifier le résultat :

JeuCompteur

```
1  import dauphine.util.*;
2  public class JeuCompteur {
3    public static void main(String[] args) {
4      Console.start();
5      int max;
6      do {
7        System.out.print("Indiquez la valeur maximale pour le choix ");
8        System.out.print("de l'ordinateur (>0) ");
9        max=Console.readInt();
10     } while(max<=0);
11     int nombre=(int)(max*Math.random());
12     int choix;
13     int nb=0;
14     do {
```

[1] Il est cependant parfois utile, dans certaines circonstances très particulières, de pouvoir passer directement d'une partie d'un programme à une autre. Comme toujours en programmation, il existe des règles générales qu'il faut parfois savoir transgresser pour améliorer le programme résultat.

[2] Comme l'ordinateur indique si le choix de l'utilisateur est plus grand ou plus petit que l'entier recherché, on peut montrer qu'il existe une stratégie optimale qui demande au plus $\lceil \log_2(n+1) \rceil$ tentatives, où n désigne le nombre d'entiers possibles et où $\lceil a \rceil$ désigne l'arrondi supérieur de a, c'est-à-dire le plus petit entier supérieur ou égal à a. On pourrait donc mesurer le score du joueur par rapport à cette stratégie optimale. Cette stratégie est d'ailleurs très simple : elle consiste à choisir un intervalle dans lequel le nombre est de façon sûre (par exemple $[0, 100]$ au départ). Le joueur propose alors le milieu de l'intervalle (50 dans l'exemple qui nous intéresse). Selon la réponse de l'ordinateur, le joueur détermine à quel demi-intervalle l'entier appartient. Il peut alors recommencer l'opération avec le demi-intervalle correct, et ceci jusqu'à trouver la solution.

```
15        System.out.print("Votre choix ? ");
16        choix=Console.readInt();
17        nb=nb+1;
18        if(choix>nombre) {
19          System.out.println("Trop grand");
20        }
21        if(choix<nombre) {
22          System.out.println("Trop petit");
23        }
24      } while (choix!=nombre);
25      System.out.println("Bravo, vous avez gagné !");
26      System.out.println("Vous avez effectué "+nb+" tentatives.");
27      System.out.println("Le nombre d'entiers possibles était "+(max+1)+".");
28    }
29  }
```

Il y a donc très peu de différences entre la version de l'exemple 5.3 et la nouvelle version. Les différences importantes apparaissent aux lignes 13, 17 et 26 du nouveau programme. La ligne 13 déclare une variable `nb` et lui donne la valeur initiale 0. La ligne 26 affiche le contenu de la variable après l'exécution de la boucle. La ligne cruciale est la ligne 17. En l'exécutant, le processeur ajoute 1 au contenu de `nb`. On comprend facilement ce qui se passe : à chaque exécution du bloc constitué des lignes 14 à 24, la valeur de `nb` augmente de 1. Comme sa valeur initiale est 0, sa valeur finale est exactement égale au nombre de fois où le bloc a été exécuté, c'est-à-dire au nombre de tours de la boucle. Ce nombre est aussi le nombre de tentatives effectuées par l'utilisateur, puisque `nb` augmente de 1 à chaque tentative. On peut donc utiliser `nb` comme mesure de la qualité du joueur.

Le nouveau comportement du programme est basé sur un mécanisme simple : on calcule la nouvelle valeur d'une variable grâce à une formule qui fait intervenir l'ancienne valeur de cette variable. On réalise ainsi un **calcul itératif**, dans lequel chaque étape nous fait progresser vers le résultat final. Le schéma peut bien sûr se compliquer en faisant intervenir plusieurs variables, et en faisant changer la formule pendant les itérations (en utilisant un mécanisme de sélection par exemple), mais le principe est toujours le même : on construit le résultat recherché étape par étape (pour d'autres exemple, cf la suite du chapitre, par exemple la section 5.4.4).

5.2.2 Technique d'analyse

Inventer des programmes basés sur des calculs itératifs n'est pas toujours facile, c'est pourquoi nous commencerons par analyser des programmes déjà écrits. Dans l'exemple de la section précédente, le calcul est tellement élémentaire qu'il est très facile de comprendre intuitivement le comportement du programme. Quand les calculs sont plus complexes, l'analyse devient plus délicate. On doit impérativement "faire tourner le programme à la main" pour le comprendre. La technique est élémentaire : il s'agit simplement de se mettre à la place du processeur et d'exécuter le programme sur le papier, en s'intéressant en particulier à l'évolution du contenu des variables.

Pour illustrer la méthode, nous allons traiter un exemple qui peu sembler assez complexe de prime abord. Voici le programme à analyser :

Calcul

```
1  import dauphine.util.*;
2  public class Calcul {
3    public static void main(String[] args) {
```

```
4     Console.start();
5     int n;
6     do {
7       n=Console.readInt();
8     } while(n<1);
9     int k=1;
10    do {
11      n/=2;
12      k+=n;
13      if(n%2==1) {
14        k/=2;
15      }
16    } while(n>0);
17    System.out.println(k);
18  }
19 }
```

Ce programme ne fait pas un calcul "sensé". On ne doit donc pas chercher ici une formule simple qui relierait la valeur initiale de `n` (c'est-à-dire la valeur choisie par l'utilisateur) à la valeur de `k` après l'exécution de la boucle (c'est-à-dire la valeur affichée par le programme). Par contre, étant donnée une valeur saisie par l'utilisateur, on doit être capable de déterminer (sans recourir à l'ordinateur) la valeur qui sera affichée.

Pour étudier le programme, nous allons tout simplement écrire l'évolution des valeurs contenues dans les variables `n` et `k` (dans un tableau). Si l'utilisateur saisit 27, nous obtenons le tableau suivant :

n	k	ligne
27		7
	1	9
13		11
	14	12
	7	14
6		11
	13	12
3		11
	16	12
	8	14
1		11
	9	12
	4	14
0		11
	4	12

Dans ce tableau, chaque ligne correspond à une modification du contenu d'une variable, provoquée par l'instruction de la ligne indiquée par la troisième colonne. Comme nous étudions une boucle, certaines instructions apparaissent plusieurs fois, ce qui est parfaitement normal. Expliquons l'évolution des variables :

- les valeurs initiales sont fixées par les hypothèses pour `n` et par la ligne 9 pour `k` ;
- lors de la première exécution de la boucle, les deux variables commencent leur évolutions : la ligne 11 place dans `n` la moitié de la valeur que cette variable contenait avant l'exécution de la ligne. Le processeur place donc 13 dans `n`. Quand la ligne 12 est exécutée, c'est bien sûr la

nouvelle valeur de `n` qui est prise en compte. De ce fait, le processeur ajoute `13` à la valeur de `k` qui contient maintenant 14. L'expression de la ligne `13` a pour valeur `true`, car `n` contient à ce moment une valeur impaire. De ce fait, le processeur exécute la ligne 14 et on se retrouve donc avec la valeur 7 dans `k` ;
- le processeur évalue l'expression du `while` et obtient la valeur `true` : il reprend donc l'exécution du bloc à la ligne 11 ;
- la nouvelle exécution de la boucle est très similaire à la première et produit elle aussi une évolution des valeurs contenues dans `n` et `k` ;
- peu à peu, le contenu de `n` décroît strictement et on finit par obtenir la valeur 0, ce qui provoque l'arrêt de la boucle. D'après le tableau d'évolution, `k` contient alors la valeur 4 qui est affichée par le processeur.

On constate que l'analyse du programme n'est pas très complexe, mais plutôt laborieuse. En effet, il est général très difficile de deviner ce que fait le programme et le seul moyen de déterminer ses effets consiste à remplacer le processeur en exécutant le programme "sur le papier". Le tableau utilisé dans l'exemple qui vient d'être étudié est un très bon outil d'analyse et peut être utilisé systématiquement pendant l'apprentissage. Nous étudierons de nombreux exemples de calculs itératifs et de constructions plus générales basées sur les boucles, et nous apprendrons ainsi à reconnaître certaines formes récurrentes. Nous pourrons ainsi nous passer d'une analyse très détaillée dans les cas les plus classiques. Mais pendant l'apprentissage et, de façon générale, quand le programme se complique, il faudra toujours avoir l'humilité de passer par une analyse précise du comportement du programme étudié.

5.2.3 Applications

Nombre de chiffres d'un entier

Considérons le problème suivant : comment déterminer le nombre de chiffres présents dans un entier donné ? On remarque que la division euclidienne par 10 d'un entier a pour effet de supprimer son chiffre le plus à droite. Par exemple, 154 devient 15. En répétant cette opération plusieurs fois, on peut supprimer tous les chiffres et obtenir 0. Par exemple, 2703 devient 270, puis 27, puis 2, puis 0. On a donc effectué 4 divisions par 10, ce qui correspond exactement au nombre de chiffres dans 2703. Pour calculer le nombre de chiffres d'un entier, on peut aussi appliquer l'algorithme suivant :

Donnée :

un entier n positif ou nul

Résultat : le nombre de chiffres de n

1. placer n dans `p`
2. initialiser `nb` à 0
3. diviser par 10 le contenu de `p`
4. ajouter 1 au contenu de `nb`
5. si `p` est strictement supérieur à 0, recommencer à l'étape 3
6. **Résultat** : le contenu de `nb`

Exécutons cet algorithme sur le papier en prenant pour n la valeur 2703. On obtient le tableau suivant d'évolution des variables :

p	nb	étape
2703		1
	0	2
270		3
	1	4
27		3
	2	4
2		3
	3	4
0		3
	4	4

L'algorithme réalise donc bien les divisions indiquées dans le raisonnement qui débute cette section. Comme la variable `nb` compte le nombre de division (elle débute à 0 et augmente de 1 après chaque division), elle contient bien à la fin de la boucle le nombre de chiffres de n. Le programme `NombreDeChiffres` donne une programmation `Java` possible pour cet algorithme.

NombreDeChiffres

```
1  import dauphine.util.*;
2  public class NombreDeChiffres {
3    public static void main(String[] args) {
4      Console.start();
5      int n;
6      do {
7        System.out.print("Choisissez un entier positif ou nul : ");
8        n=Console.readInt();
9      } while (n<0);
10     int nb=0;
11     int p=n;
12     do {
13       p/=10;
14       nb++;
15     } while(p>0);
16     System.out.println(n+" possède "+nb+" chiffres(s)");
17   }
18 }
```

Analyse d'un programme

Voici maintenant un exercice corrigé qui demande d'analyser un programme :

Exercice corrigé

On donne le programme suivant :

Exercice

```
1  import dauphine.util.*;
2  public class Exercice {
3    public static void main(String[] args) {
4      Console.start();
5      int n;
6      do {
7        System.out.print("Choisissez un entier positif ou nul : ");
```

```
8         n=Console.readInt();
9       } while (n<0);
10      int s=0;
11      do {
12        s+=n%10;
13        n/=10;
14      } while(n>0);
15      System.out.println(s);
16    }
17  }
```

Questions :

1. *Donnez l'évolution des variables pendant l'exécution du programme si l'utilisateur saisit l'entier* 3728.
2. *Décrivez simplement le lien entre l'affichage final du programme et l'entier saisi par l'utilisateur en justifiant votre réponse.*

La première question est classique et on peut y répondre en faisant tourner le programme sur le papier. On obtient le résultat suivant :

`n`	`s`	ligne
3728		8
	0	10
	8	12
372		13
	10	12
37		13
	17	12
3		13
	20	12
0		13

La valeur finale de `s` est donc 20.

Pour répondre à la deuxième question, il faut étudier attentivement les lignes 12 et 13. Dans la ligne 12, on ajoute au contenu de `s` la valeur du reste de la division de `n` par 10. Or, on constate que ce reste est le chiffre le plus à droite de `n`. Comme la ligne 13 a pour effet de supprimer ce chiffre, on va ajouter à `s` tous les chiffres de l'entier saisi au départ, de droite à gauche. La valeur affichée à la fin du programme est donc la somme des chiffres de l'entier saisi par l'utilisateur.

5.3 Boucle conditionnelle pré-testée

5.3.1 Motivation

Le seul inconvénient de la boucle `do while` est que l'instruction répétée est exécutée au moins une fois, quelle que soit la valeur de l'expression booléenne (rappelons que la boucle est *post* testée). Dans certains cas, ce n'est pas une très bonne idée, par exemple quand le traitement à répéter n'est pas toujours applicable ou utile. Si nous reprenons l'exemple du programme `Exercice` (l'exercice corrigé de la section précédente, 111), on remarque que quand l'entier saisi par l'utilisateur est 0, on effectue tout de même les calculs (quotient et reste de la division par 10) alors qu'ils sont inutiles :

la somme des chiffres est bien sûr 0. On pourrait remédier à cet inconvénient si on pouvait placer le `while` au début de la boucle plutôt qu'à la fin.

5.3.2 L'instruction `while`

Forme générale

On cherche donc à réaliser une boucle conditionnelle pré-testée, c'est-à-dire testée *avant* l'exécution, ce que permet l'instruction `while`, dont la forme générale est la suivante :

```
while (expression de type boolean)
  instruction;
```

Comme pour toutes les instructions composées, il est vivement conseillé de toujours passer par un bloc, ce qui donne la forme générale suivante :

```
while (expression de type boolean) {
  instruction 1;
  ...
  instruction n;
}
```

Sémantique

L'interprétation par le processeur d'une instruction `while` est très simple :

1. le processeur évalue l'expression qui suit le `while`
2. si le résultat est `false`, l'exécution de l'instruction `while` est terminée
3. sinon :
   (a) le processeur exécute l'instruction qui suit l'expression (ou le bloc, le cas échéant)
   (b) le processeur reprend l'exécution à l'étape 1

La différence avec le `do while` est très claire : pour un `while`, l'expression booléenne est évaluée **avant** l'exécution (éventuelle) du bloc à répéter, alors que dans un `do while`, l'expression est évaluée **après** l'exécution du bloc. Il est donc possible que le bloc d'un `while` ne soit jamais exécuté, alors qu'avec un `do while`, il est toujours exécuté au moins une fois.

Application

En utilisant un `while` nous pouvons rendre plus efficace le programme qui calcule la somme des chiffres d'un entier quelconque, comme l'illustre l'exemple suivant :

Exemple 5.5 :

On propose la solution suivante :

SommeDesChiffres

```
1  import dauphine.util.*;
2  public class SommeDesChiffres {
3    public static void main(String[] args) {
4      Console.start();
5      int n;
6      do {
7        System.out.print("Choisissez un entier positif ou nul : ");
8        n=Console.readInt();
```

```
9       } while (n<0);
10      int s=0;
11      while(n>0) {
12        s+=n%10;
13        n/=10;
14      }
15      System.out.println(s);
16    }
17  }
```

L'intérêt de cette version réside dans l'utilisation du `while`. Quand l'utilisateur saisit un 0, la boucle `while` ne s'exécute pas, car la somme des chiffres est bien entendu 0. Dans la version avec `do while`, le processeur faisait des calculs inutiles. Quand l'utilisateur saisit un nombre strictement positif, les calculs sont effectués et tout se passe alors d'une façon très similaire à celle de la solution avec un `do while`. Il suffit d'analyser quelques exemples pour s'en convaincre[3].

5.3.3 Algorithme et organigramme

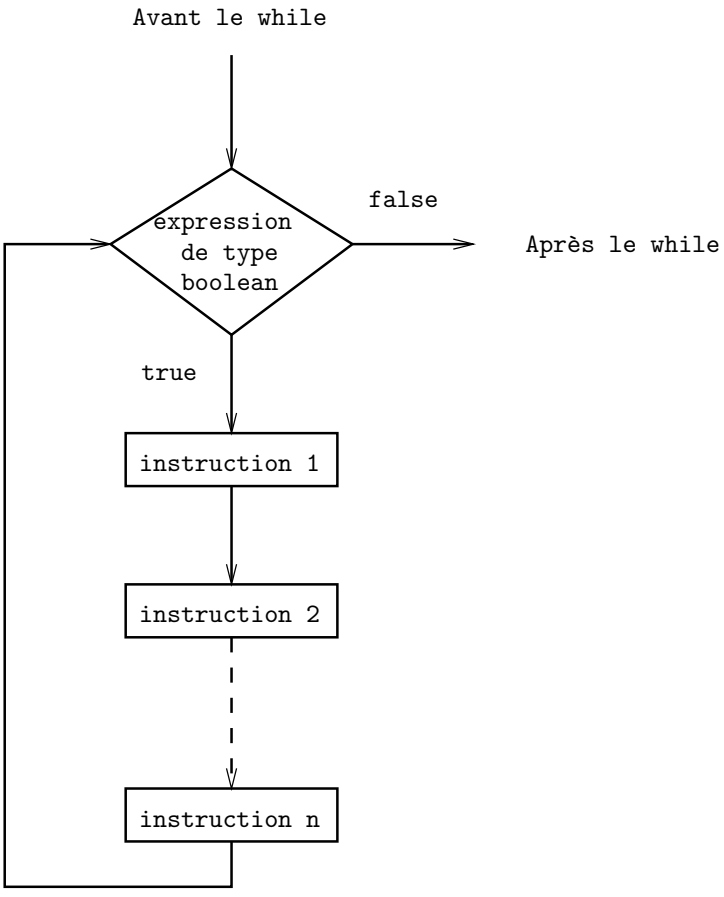

FIG. 5.3 – Organigramme d'un `while`

Les organigrammes des programmes ou algorithmes utilisant un `while` ne posent aucun problème particulier, la sémantique du `while` étant très proche de celle du `do while`. La figure 5.3

[3] Tâche lâchement laissée au lecteur à titre d'exercice...

illustre cette sémantique (voir la figure 5.4 pour un exemple d'application du `while`). Pour les algorithmes, on préfère en général éviter la description que nous avons utilisée pour donner l'interprétation par le processeur de l'instruction. Considérons par exemple le calcul de la somme des chiffres d'un nombre (exemple 5.5). Voici un algorithme possible :

Donnée :
un entier $n \geq 0$

Résultat : la somme des chiffres de n

1. initialiser `s` à zéro
2. initialiser `p` à n
3. tant que `p` est strictement supérieur à 0 :
   (a) ajouter à `s` le reste de la division de `p` par 10
   (b) diviser par 10 le contenu de `p`
4. **Résultat** : le contenu de `s`

L'algorithme utilise donc la traduction française de la construction `while`. Voici un autre exemple :

Exemple 5.6 :

On donne l'algorithme suivant :

Donnée :
un entier $n \geq 1$

Résultat : le plus petit entier p tel que $2^p \geq n$, obtenu sans utiliser les fonctions ln et exp.

1. initialiser `puissance` à 1
2. initialiser `p` à 0
3. tant que `puissance` est strictement inférieure à n :
   (a) multiplier par 2 le contenu de `puissance`
   (b) ajouter 1 à `p`
4. **Résultat** : le contenu de `p`

En exécutant sur le papier cet algorithme, on peut se convaincre qu'il réalise bien la tâche prévue. Donnons par exemple le tableau d'évolution du contenu des variables pour $n = 57$:

puissance	p	étape
1		1
	0	2
2	0	3.1
	1	3.2
4	0	3.1
	2	3.2
8	0	3.1
	3	3.2
16	0	3.1
	4	3.2
32	0	3.1
	5	3.2
64	0	3.1
	6	3.2

Le résultat est donc 6. On remarque qu'après l'exécution des deux étapes de la boucle, le contenu de `puissance` est toujours 2^p. C'est cette propriété qui assure le fonctionnement de l'algorithme.

Voici une version `Java` de l'algorithme :

PuissanceDeDeux

```
1  import dauphine.util.*;
2  public class PuissanceDeDeux {
3    public static void main(String[] args) {
4      Console.start();
5      int n;
6      do {
7        System.out.print("Choisissez un entier supérieur ou égal à 1 : ");
8        n=Console.readInt();
9      } while (n<1);
10     int puissance=1;
11     int p=0;
12     while(puissance<n) {
13       puissance*=2;
14       p++;
15     }
16     System.out.println(p+" est le plus petit entier p tel que 2^p="
17                        +puissance+" soit supérieur ou égal à "+n);
18   }
19 }
```

La figure 5.4 donne l'organigramme de la solution `Java` du problème.

5.4 Boucle non conditionnelle

5.4.1 Exemple introductif

Il est souvent utile de pouvoir faire répéter des opérations par l'ordinateur sans être dans le cadre d'une boucle conditionnelle. Pour commencer par un exemple simple, on peut vouloir calculer k^n, où k et n sont des entiers strictement positifs proposés par l'utilisateur. On pourrait bien sûr calculer k^n en utilisant la méthode `pow` de la classe `Math` (cf le chapitre 3), mais le calcul serait alors fait en `double` et serait approximatif (nous allons travailler ici avec des `long`s). Par définition, k^n est obtenu en multipliant n fois par k la valeur 1. On doit donc répéter n fois une opération simple de multiplication, selon l'algorithme suivant :

Données :
- k un entier strictement positif
- n un entier positif

Résultat : k^n

1. initialiser `puissance` à 1
2. répéter n fois :
   (a) multiplier par k le contenu de `puissance`
3. **Résultat** : le contenu de `puissance`

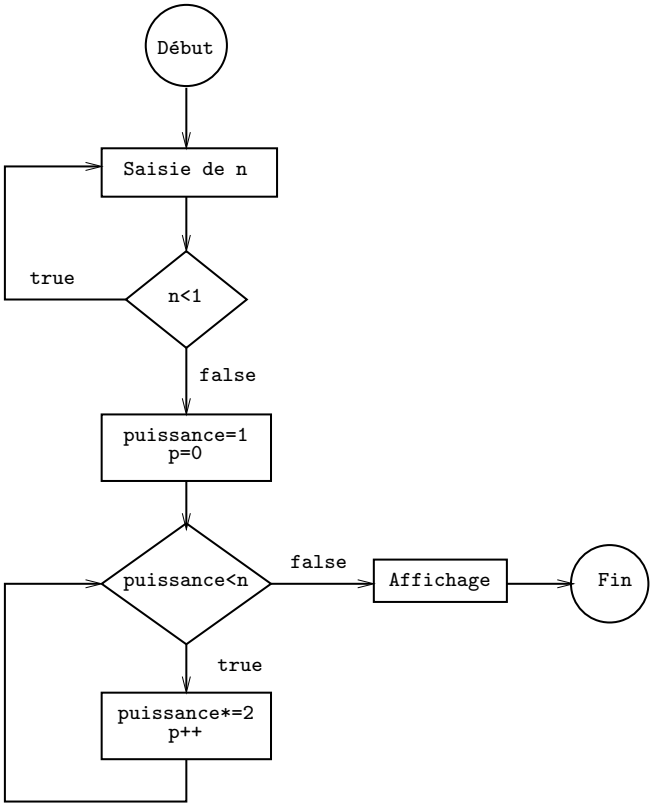

FIG. 5.4 – Organigramme du programme `PuissanceDeDeux`

L'algorithme obtenu est très simple, mais pose un problème : nous ne connaissons pas de construction permettant de répéter n fois une instruction. Pour traduire cet algorithme en `Java` nous devons donc utiliser une boucle conditionnelle, et compter le nombre d'exécution de cette boucle, comme dans le programme `PuissanceDeDeux` de la section 5.3.3. Nous pouvons par exemple proposer le programme suivant :

PuissanceWhile

```
1  import dauphine.util.*;
2  public class PuissanceWhile {
3    public static void main(String[] args) {
4      Console.start();
5      long k=Console.readLong();
6      long n=Console.readLong();
7      long puissance=1;
8      long p=0;
9      while(p<n) {
10       puissance*=k;
11       p++;
12     }
13     System.out.println(k+"^"+n+"="+puissance);
14   }
15 }
```

Pour simplifier le programme, nous n'avons pas proposé de saisie contrôlée. De ce fait, le programme peut très bien ne pas donner un résultat très pertinent si l'utilisateur ne saisit pas des entiers

convenables. Le but de cette présentation est bien entendu de se focaliser sur la répétition en elle-même, pas sur les saisies.

Comme les répétitions non conditionnelles sont assez fréquentes, `Java` propose une instruction particulière, la boucle `for`, qui simplifie leur présentation. En utilisant un `for`, le programme précédent devient :

PuissanceFor

```
1  import dauphine.util.*;
2  public class PuissanceFor {
3    public static void main(String[] args) {
4      Console.start();
5      long k=Console.readLong();
6      long n=Console.readLong();
7      long puissance=1;
8      for(long p=0;p<n;p++) {
9        puissance*=k;
10     }
11     System.out.println(k+"^"+n+"="+puissance);
12   }
13 }
```

Il y a peu de différences entre les deux programmes, tout résidant dans le remplacement de la boucle `while` par une boucle `for`. Les deux programmes fonctionnent exactement de la même manière.

5.4.2 L'instruction `for`

Une forme générale

La forme générale du `for` étant assez complexe, nous nous contenterons dans un premier temps d'une version simplifiée :

`for(`*type entier compteur=valeur initiale*`;`*compteur*`<=`*valeur finale*`;`*compteur*`++)`
  instruction`;`

Comme toujours, il est préférable de toujours utiliser un bloc, ce qui donne :

`for(`*type entier compteur=valeur initiale*`;`*compteur*`<=`*valeur finale*`;`*compteur*`++) {`
  instruction 1`;`
  ...
  instruction n`;`
`}`

Diverses variantes sont utilisables. On peut par exemple remplacer

compteur`<=`*valeur finale*

par

compteur`<`*valeur finale*

On peut aussi remplacer

`for(`*type entier compteur=valeur initiale*`;`*compteur*`<=`*valeur finale*`;`*compteur*`++)`

par

`for(`*type entier compteur=valeur initiale*`;`*compteur*`>`*valeur finale*`;`*compteur*`--)`

Dans cette dernière version, on peut aussi remplacer le `>` par un `>=`. Nous verrons de façon générale qu'il existe de nombreuses autres possibilités (voir la section 5.5.3).

REMARQUE

En fait, il n'est pas obligatoire d'utiliser un type entier. On pourrait en effet utiliser un type réel, comme l'illustre l'exemple 5.15. Pour la forme simplifiée du `for`, il est cependant préférable de se limiter aux types entiers.

Sémantique

Il y a plusieurs façons de considérer la sémantique du `for`. On peut d'accord commencer par remplacer le `for` par sa traduction sous forme d'un `while`. On peut considérer que l'instruction composée suivante

```
for(type entier compteur=valeur initiale;compteur<=valeur finale;compteur++) {
  instruction 1;
  ...
  instruction n;
}
```

est approximativement équivalente à la boucle `while` suivante[4] :

```
{
  type entier compteur=valeur initiale;
  while(compteur<=valeur finale) {
    instruction 1;
    ...
    instruction n;
    compteur++;
  }
}
```

La sémantique du `for` se déduit alors facilement de celle du `while`. La seule astuce réside dans le fait que l'ensemble de l'instruction est placée dans **un bloc**, ce qui signifie que **la variable `compteur` n'est pas utilisable en dehors de l'instruction composée** (voir à ce sujet la section 5.5.1). Voici donc la sémantique du `for` :

1. le processeur crée la variable *`compteur`* et lui affecte la *`valeur initiale`*
2. le processeur évalue l'expression *`compteur<=valeur finale`*
3. si la valeur obtenue est `false`, l'exécution du `for` est terminée
4. sinon :
   (a) le processeur exécute les instructions *`1`* à *`n`*
   (b) le processeur exécute l'instruction *`compteur++`*
   (c) le processeur retourne à l'étape 2

On obtient donc un `while` en plus complexe ! Fort heureusement, on peut déduire de cette description une version plus simple. On remarque en effet que le processeur exécute le corps du `for` en faisant prendre *successivement* à la variable *`compteur`* toutes les valeurs entières comprises entre

[4] En fait, pas exactement, car il existe une différence assez subtile entre les deux boucles. Cette différence concerne un aspect des boucles largement hors programme et, dans ce document, on considérera que les deux boucles sont équivalentes. Le lecteur intéressé par les détails techniques pourra se reporter à [7], en particulier la section 6.7.3 qui contient une discussion sur les différences entre le `for` et le `while`. Notons en résumé que tant qu'on évite d'utiliser l'instruction `continue` les deux boucles sont strictement équivalentes.

valeur initiale et *valeur finale*. On obtient donc *valeur finale - valeur initiale +1* exécutions de la boucle, ce qui correspond exactement à une répétition non conditionnelle. La sémantique de la boucle est donc la suivante :

1. pour tous les entiers compris au sens large entre *valeur initiale* et *valeur finale* (dans l'ordre croissant) :
   (a) le processeur donne à *compteur* la valeur de l'entier considéré
   (b) le processeur exécute les instructions *1* à *n*

REMARQUE

Cette vision simplifiée du `for` permet de comprendre rapidement un programme utilisant une telle boucle. Elle est pourtant trop simplifiée et seule l'équivalence avec le `while` permet de bien comprendre le comportement d'une boucle `for` quelconque. Nous verrons en effet que la forme générale d'un `for` se traduit assez simplement en un `while`, se qui permet de comprendre son fonctionnement (voir la section 5.5.3). La sémantique simplifiée que nous venons de voir ne s'applique qu'à un cas très particulier d'utilisation du `for`. Si on modifie la condition *compteur*`<=`*valeur finale* en remplaçant le `<=` par un `<`, il faut changer de sémantique car la *valeur finale* n'est maintenant plus atteinte.

On peut noter au passage que désigner la boucle `for` comme une boucle "non conditionnelle" est en fait incorrect : un `for` est en dernière analyse une sorte de `while`, c'est-à-dire une boucle conditionnelle. Seule la version simplifiée du `for` peut être considérée comme non conditionnelle car le nombre d'itérations de la boucle ne dépend pas des calculs qu'elle effectue, mais plus simplement des valeurs extrêmes choisies pour le compteur.

Exemple 5.7 :

Voici un programme très simple utilisant des boucles `for` :

ForAffiche

```
1  public class ForAffiche {
2    public static void main(String[] args) {
3      for(int i=0;i<5;i++) {
4        System.out.println(i);
5      }
6      for(int j=6;j>=0;j--) {
7        System.out.println(j);
8      }
9    }
10 }
```

Ce programme ne fait rien de bien intéressant, il se contente d'illustrer le `for`. Il affiche :

AFFICHAGE

```
0
1
2
3
4
6
5
4
```

```
3
2
1
0
```

L'affichage des entiers croissants de 0 à 4 correspond à la première boucle (lignes 3 à 5). Le processeur commence par donner la valeur 0 à `i`. Il vérifie que cette valeur est strictement inférieure à 5, puis exécute la ligne 4, ce qui provoque l'affichage de 0. Il incrémente alors `i` de 1, et recommence la boucle. Il est facile de comprendre pourquoi la valeur 5 n'est pas affichée : la condition du `for` est `i<5`, donc quand la variable `i` atteint la valeur 5, la boucle s'arrête.

La second partie de l'affichage (entiers décroissants de 6 à 0) correspond au deuxième `for` (lignes 6 à 8). Son analyse est élémentaire, à condition encore une fois de penser au `while` équivalent. La variable `j` commence à la valeur 6, et à chaque itération de la boucle, l'instruction `j-` la décrémente de 1. Ceci explique pourquoi les entiers décroissent de 6 à 0. Lors de la dernière exécution de la boucle, la variable `j` prend la valeur -1, ce qui provoque l'arrêt de la boucle, à cause de la condition `j>=0`.

5.4.3 Une application

Nous avons déjà vu comment utiliser un `for` pour calculer k^n. Voyons maintenant comment calculer $n!$. Notons tout d'abord qu'aucune méthode déjà existante en `Java` ne propose ce résultat. Il est de ce fait **obligatoire** d'écrire une boucle pour calculer $n!$. Pour obtenir $n!$, il faut par définition multiplier 1 par 2, puis le résultat par 3, par 4, etc. jusqu'à $n-1$ et n, car $n! = 1\times2\times\cdots(n-1)\times n$. Il y a donc une opération répétitive (la multiplication) qu'il faut effectuer $n-1$ fois. Plus précisément, on doit multiplier le résultat précédent par i pour i allant de 2 à n. On est donc exactement dans le cadre d'une boucle `for` et on peut appliquer l'algorithme suivant :

Donnée :

n un entier positif

Résultat : $n!$

1. initialiser `fact` à 1
2. pour `i` allant de 2 à n :
   (a) multiplier le contenu de `fact` par `i`
3. **Résultat :** le contenu de `fact`

L'algorithme utilisé est différent de ceux étudiés précédemment. Il est beaucoup plus proche d'un programme `Java` : c'est souvent le cas quand on utilise une boucle `for`, car on doit en général préciser qu'on utilise un compteur (`i` dans cet exemple) et indiquer ses valeurs initiale et finale. Voici une traduction possible de cet algorithme :

Factorielle

```
1 import dauphine.util.*;
2 public class Factorielle {
3   public static void main(String[] args) {
4     Console.start();
5     int n=Console.readInt();
6     long fact=1;
7     for(int i=2;i<=n;i++)
8       fact*=i;
```

```
9       System.out.println(n+"!="+fact);
10    }
11  }
```

On remarque l'emploi des types `int` et `long`. En effet, la fonction factorielle croit très rapidement. Même pour de petites valeurs de n, $n!$ dépasse les capacités des `int`s (12! est un `int`, mais ce n'est plus le cas de 13!). C'est pourquoi on utilise un `long` pour contenir le résultat du calcul. On pourrait bien sûr utiliser un `byte` pour contenir n, mais ce type est moins facile à utiliser que `int`. Notons que même les `long`s ne permettent pas d'aller très loin : 20! est un `long`, mais ce n'est pas le cas de 21!. Comme les calculs en `long` sont inexacts, le résultat obtenu pour 21! sera totalement faux (revoir à ce sujet le chapitre 2 et en particulier la section 2.4.2).

5.4.4 Schéma algorithmique : mise d'un calcul sous forme récurrente

Introduction

L'application proposée dans la section précédente peut sembler anodine. Ce n'est pas le cas, car elle constitue un exemple d'une forme très générale d'utilisation de la boucle `for` pour calculer des résultats numériques.

Pour bien comprendre le schéma général, il faut utiliser un langage mathématique, celui des suites. Revenons en effet à l'exemple de la factorielle. La définition par ellipse ($n! = 1 \times 2 \times \cdots (n-1) \times n$) n'est pas entièrement satisfaisante mathématiquement. On lui préfère une définition itérative (par récurrence) sous la forme suivante :

$$\begin{cases} 1! = 1 \\ n! = n \times (n-1)! \text{ pour } n > 1 \end{cases}$$

Cette définition est très simple à comprendre : pour calculer $n!$, il suffit de multiplier $(n-1)!$ par n. Or, pour obtenir $(n-1)!$, il suffit de multiplier $(n-2)!$ par $(n-1)$, etc. On finit par atteindre 1! qui par définition vaut 1. Alors $2! = 2 \times 1$, puis $3! = 3 \times 2! = 3 \times 2 \times 1$, etc. Le gros avantage de la définition itérative est qu'elle se traduit instantanément en l'algorithme proposé à la section précédente. Le principe est que quand le compteur de la boucle `for` vaut k, la boucle est en train de calculer $k!$.

Principe général

Pour appliquer l'algorithme précédent à d'autres calculs, il suffit de rendre sa formulation plus abstraite. On souhaite maintenant calculer le terme numéro n de la suite $(u_p)_{p\in\mathbb{N}}$. On suppose que cette suite est définie par une récurrence d'ordre 1, c'est-à-dire qu'on calcule u_k en fonction de u_{k-1} et de k. Pour fixer les idées, on écrit $u_k = f(k, u_{k-1})$. Dans le cas de la fonction factorielle, on pose $u_k = k!$, ce qui donne $u_k = k u_{k-1}$. L'algorithme suivant permet le calcul de u_n :

Donnée :

n un entier positif

Résultat : u_n

1. initialiser `val` à u_0
2. pour `i` allant de 1 à n :
   (a) placer dans `val` le résultat de $f($`i`$,$ `val`$)$
3. **Résultat :** le contenu de `val`

Pour bien comprendre le fonctionnement de cet algorithme, le plus simple est d'analyser son fonctionnement sur un exemple. Avec $n = 3$, on obtient le tableau suivant pour l'évolution des variables :

`val`	`i`	étape
u_0		1
	1	2
$f(1, u_0) = u_1$		2.1
	2	2
$f(2, u_1) = u_2$		2.1
	3	2
$f(3, u_2) = u_3$		2.1
	4	2

Commençons par une remarque : nous indiquons que l'évolution du compteur (`i`) se fait à l'étape 2, afin d'éviter toute confusion. La valeur 4 indiquée sur la dernière ligne correspond au comportement classique d'un `for` : avant de terminer la boucle, une dernière mise à jour est effectuée. Le compteur dépasse alors la valeur finale, ce qui provoque l'arrêt de la boucle.

L'évolution du contenu de `val` est très claire : avant le démarrage de la boucle, `val` contient par définition u_0. Ensuite, à la fin d'une exécution du corps de la boucle (qui se limite à l'étape 2.1 ici), le contenu de `val` est toujours $u_{\texttt{i}}$. Ceci permet à l'exécution suivante de la boucle d'être dans les bonnes conditions pour appliquer la formule de récurrence.

Le schéma algorithmique est entièrement basé sur le fait que le calcul envisagé peut s'écrire comme le terme d'une suite définie par récurrence. Dès qu'on peut mettre un calcul sous forme récurrente, on peut utiliser une simple boucle `for` pour le mettre en œuvre, et c'est en général le moyen le plus simple de le faire.

Exemple d'utilisation

Considérons le calcul $1 + 2 + \cdots + n$ qui s'écrit plus formellement $\sum_{i=1}^{n} i$. Nous connaissons bien sûr une simplification de cette formule qui permet de la calculer de façon directe, mais nous allons essayer de faire le calcul sans utiliser la forme simplifiée. Il est bien entendu impossible d'écrire directement $1 + 2 + \cdots + n$ dans un programme. Il nous faut donc mettre en place un algorithme permettent de faire ce calcul. Or, on remarque qu'il s'agit simplement d'ajouter 2 à 1, puis 3 au résultat, puis 4 au nouveau résultat, etc. jusqu'à atteindre n. Appelons $u_k = 1 + 2 + \cdots + k$. Il est clair que $u_k = k + u_{k-1}$ et on peut même poser $u_0 = 0$. On est donc exactement dans un cas particulier du schéma général proposé dans ce qui précède. On peut donc appliquer directement l'algorithme proposé et on obtient le programme suivant :

Somme

```
1  import dauphine.util.*;
2  public class Somme {
3    public static void main(String[] args) {
4      Console.start();
5      int n=Console.readInt();
6      int val=0;
7      for(int i=1;i<=n;i++) {
8        val+=i; // forme simplifiée de val=val+i
9      }
10     System.out.println(val);
11   }
12 }
```

Importance

Le schéma algorithmique proposé est crucial car c'est le moyen le plus simple (et souvent le seul) pour calculer certaines grandeurs définies par une formule de récurrence. Il existe parfois des formes simplifiées (c'est le cas pour la somme des entiers par exemple) mais en général ce n'est pas le cas (comme pour la factorielle, par exemple). De plus, les formes simplifiées sont parfois moins précises, comme quand on calcule k^n par exemple (voir la section 5.4.1). Notons au sujet de ce calcul qu'il s'agit bien d'une application du schéma général. En effet, on peut définir la suite $u_p = k^p$. Elle vérifie la relation de récurrence $u_p = ku_{p-1}$ avec $u_0 = 1$. La formule de récurrence est plus simple que celles étudiées dans les exemples précédents car elle ne fait pas apparaître directement la valeur de p, mais il s'agit tout de même d'une application du schéma général.

Voici maintenant un exemple moins mathématique, mais plus proche de la pratique :

Exemple 5.8 :

On souhaite calculer la moyenne de notes entrées par l'utilisateur. Pour ce faire, on va d'abord calculer la somme des notes puis on divisera cette somme par le nombre de notes. Pour calculer la somme, on utilise le schéma récurrent classique. Le seul problème est que *les notes doivent être saisies au fur et à mesure de l'exécution de la boucle*. En effet, nous ne connaissons pas de moyen de stocker plusieurs valeurs dans une "variable" dont la capacité serait déterminée au moment de l'exécution du programme[5]. A chaque exécution de la boucle, on doit demander une nouvelle note à l'utilisateur puis ajouter cette note à la somme courante afin d'obtenir la nouvelle somme. Voici l'algorithme correspondant :

Données :
– n le nombre de notes

Résultat : la moyenne des notes.

1. initialiser `somme` à 0.
2. répéter n fois les instructions suivantes :
   (a) saisir une note
   (b) ajouter cette note à `somme` et placer le résultat dans `somme`
3. Résultat : `somme` divisé par n

Voici maintenant le programme correspondant :

MoyenneDeNNotes

```
1  import dauphine.util.*;
2  public class MoyenneDeNNotes {
3    public static void main(String[] args) {
4      Console.start();
5      System.out.println("Nombre de notes ?");
6      int n=Console.readInt();
7      double moyenne=0,note;
8      for(int k=1;k<=n;k++) {
9        System.out.println("Note numéro "+k);
10       note=Console.readDouble();
11       moyenne=moyenne+note;
12     }
13     System.out.println("Moyenne : "+moyenne/n);
14   }
15 }
```

[5] De telles "variables" existent, nous les étudierons au chapitre 8.

Extension à une récurrence d'ordre supérieur

Le schéma algorithmique proposé ne s'applique qu'à la récurrence d'ordre 1. Considérons l'exemple de la suite de Fibonacci définie par la récurrence d'ordre 2 suivante :

$$\begin{cases} u_1 = 1 \\ u_2 = 1 \\ u_n = u_{n-1} + u_{n-2} \text{ pour } n > 2 \end{cases}$$

Pour calculer u_n, il faut connaître les valeurs de u_{n-1} et de u_{n-2}. Il faut donc disposer d'au moins deux variables qui contiendront ces valeurs. Le plus simple est de placer u_n dans une troisième variable. On veut donc répéter un calcul de la forme `c=b+a`, où `a` contient u_{n-2}, `b` u_{n-1} et `c` u_n. La difficulté vient de la répétition. Si on démarre la boucle avec `a=1` (c'est-à-dire u_1) et `b=1` (c'est-à-dire u_2), après le calcul `c=b+a`, la variable `c` contient u_3. Si on souhaite calculer u_4, il faut que le calcul soit effectué une nouvelle fois. Mais pour cette nouvelle itération, il faudra que `a` contienne u_2 et que `b` contienne u_3. Il faut donc compléter le calcul lui-même par une mise à jour de `a` et `b` qui jouent le rôle de "mémoire" pour la boucle. Voici donc un algorithme possible :

Donnée :

n un entier strictement positif

Résultat : le terme u_n de la suite de Fibonacci

1. initialiser `a`, `b` et `c` à 1
2. répéter $n-2$ fois :
   (a) placer dans `c` le résultat de `a+b`
   (b) placer dans `a` le contenu de `b`
   (c) placer dans `b` le contenu de `c`
3. **Résultat :** le contenu de `c`

Toute la subtilité de l'algorithme vient du fait qu'une itération de la boucle fait deux choses :

1. elle calcule le terme voulu de la suite ;
2. elle prépare l'itération suivante.

La variable `c` est initialisée à 1 afin de traiter les cas où n vaut 1 ou 2.

Il est facile de traduire cet algorithme en un programme, comme par exemple :

Fibonacci

```
1  import dauphine.util.*;
2  public class Fibonacci {
3    public static void main(String[] args) {
4      Console.start();
5      int n;
6      do {
7        n=Console.readInt();
8      } while(n<1);
9      int a=1,b=1,c=1;
10     for(int i=2;i<=n;i++) {
11       c=a+b;
12       a=b;
13       b=c;
14     }
```

```
15      System.out.println(c);
16    }
17  }
```

Il est intéressant d'analyser le comportement du programme sur un exemple simple en donnant un tableau d'évolution des variables, en supposant que l'utilisateur choisisse la valeur 5 :

a	b	c	i	ligne
1	1	1		9
			2	10
		2		11
1				12
	2			13
			3	10
		3		11
2				12
	3			13
			4	10
		5		11
3				12
	5			13
			5	10
		8		11
5				12
	8			13
			6	10

On constate que la variable `a` prend les valeurs de u_n pour n allant de 1 à 4, alors que `b` et `c` prennent les valeurs de u_n pour n allant de 1 à 5. On voit aussi comment les lignes 12 et 13 préparent l'itération suivante de la boucle.

REMARQUE

L'exemple qui vient d'être traité illustre et applique une règle générale à retenir ; une itération d'une boucle doit faire deux choses :

1. traiter les éléments concernés par l'itération en cours (par exemple calculer le terme voulu de la suite que la boucle calcule) ;
2. préparer l'itération suivante.

La deuxième étape est celle qui assure en général le fonctionnement du programme. En effet, la première étape n'est possible que grâce à certaines conditions. Dans l'exemple de la suite de Fibonacci, il faut que les variables `a` et `b` contiennent les deux dernières valeurs de la suite. Après avoir réalisée la première étape, les conditions ne sont en général pas vérifiées pour l'itération suivante de la boucle. Le rôle de la deuxième étape est justement de faire en sorte de ces conditions soient vérifiées.

5.4.5 Intérêt du `for`

Comme un `for` est équivalent à un `while`, on peut se demander si cette nouvelle instruction est vraiment utile. On peut dire que la situation est assez proche de celle du `switch` comparé au `if else` : le principal intérêt du `for` est d'améliorer la lisibilité des programmes.

Comme nous le verrons à la section 5.5.3, tout ce qui est réalisable par un `while` l'est par un `for`. Cependant, les programmeurs utilisent en général le `for` selon la forme restreinte que nous avons étudiée dans la présente section. Le gros avantage de cette forme restreinte réside justement dans ses restrictions : d'un seul coup d'oeil, le programmeur expérimenté comprend ce que fait la boucle. Il détermine l'intervalle des valeurs prises par le compteur, si les bornes sont prises en compte, le sens de variation du compteur, etc. Comme dans le cas du `switch`, le comportement global du programme peut être analysé rapidement.

Il est bien sûr possible d'écrire des boucles `for` qui vont induire le programmeur en erreur, essentiellement en utilisant une forme complexe. Mais le but de la programmation n'est pas seulement d'obtenir un programme qui fonctionne : c'est aussi d'obtenir un programme facile à maintenir, c'est-à-dire facile à modifier (pour corriger les *bugs*, pour ajouter de nouvelles fonctions, etc.). Or, les programmes professionnels sont volumineux et il est très rare qu'un programmeur travaille seul, et quasiment impossible qu'un programmeur connaisse chaque ligne de programme par cœur, d'autant plus que la durée de vie d'un programme est souvent supérieure à la durée de présence d'un programmeur dans une entreprise. Il faut donc impérativement qu'on puisse relire facilement un morceau de programme et comprendre rapidement son fonctionnement. L'instruction `for`, utilisée à bon escient, permet d'aller dans cette direction. Cela ne veut par dire que l'instruction `while` est à bannir, au contraire. Il faut simplement réserver le `while` aux boucles conditionnelles, alors que le `for` sera utilisé pour les boucles non conditionnelles (plus précisément pour les boucles dont le nombre d'itérations ne dépend pas des opérations effectuées à chaque tour).

REMARQUE

Comme l'ont montré les nombreux exemples de cette section, il n'est absolument pas obligatoire d'utiliser le compteur d'une boucle `for` dans le corps de celle-ci. L'intérêt du `for` est d'indiquer clairement qu'on connaît le nombre d'itérations que va réaliser une boucle avant de l'exécuter. Dans certaines applications, les valeurs prises par le compteur sont importantes, dans d'autres, seul le nombre d'exécutions l'est.

5.5 Subtilités des boucles

5.5.1 Déclaration de variables

Nous avons vu à la section 4.2.2 qu'une déclaration de variable possède une portée, ce qui peut parfois poser des problèmes, par exemple quand on utilise un `if`. Comme nous allons le voir dans la présente section, il faut aussi être attentif quand on utilise des boucles.

Commençons par un exemple simple qui correspond à une erreur classique :

Exemple 5.9 :

On souhaite que l'utilisateur saisisse un entier non nul et on propose le programme suivant :

NonNulFaux

```
1  import dauphine.util.*;
2  public class NonNulFaux {
3    public static void main(String[] args) {
4      Console.start();
5      do {
6        System.out.print("Choisissez un entier non nul : ");
7        int j=Console.readInt();
8      } while (j==0);
9      System.out.println(j);
```

```
10    }
11  }
```

Le compilateur refuse le programme et indique les erreurs suivantes :

ERREUR DE COMPILATION

```
NonNulFaux.java:8: cannot resolve symbol
symbol  : variable j
location: class NonNulFaux
    } while (j==0);
             ^
NonNulFaux.java:9: cannot resolve symbol
symbol  : variable j
location: class NonNulFaux
    System.out.println(j);
                       ^
2 errors
```

En fait, le compilateur considère que la variable `j` n'existe pas quand il compile les lignes 8 et 9. Ceci est parfaitement normal : la variable `j` est déclarée à la ligne 7, dans un bloc. D'après la section 4.2.2 la portée de `j` est donc constituée uniquement du bloc qui contient sa déclaration. Comme les lignes 8 et 9 ne sont pas dans ce bloc, elles ne peuvent pas utiliser la variable `j`.

Comme on utilise toujours des blocs dans les boucles, il est très fréquent de rencontrer le problème qu'illustre l'exemple précédent. Dans la pratique, cela signifie qu'il faut être très attentif aux déclarations de variables contenues dans une boucle : elles sont parfois utiles, mais il faut impérativement être sûr de ne pas avoir besoin des variables ainsi déclarées en dehors de la boucle. Ce point est parfois mal compris : il faut bien noter que l'expression booléenne qui termine un `do while` **ne fait pas partie du bloc qui constitue le corps de la boucle**.

REMARQUE

On pourrait croire résoudre le problème de l'exemple précédent en utilisant une boucle simplifiée, sans bloc, comme par exemple :

NonNulToujoursFaux

```
1  import dauphine.util.*;
2  public class NonNulToujoursFaux {
3    public static void main(String[] args) {
4      Console.start();
5      do
6        int j=Console.readInt();
7      while (j==0);
8      System.out.println(j);
9    }
10 }
```

Pour des raisons techniques qui dépassent le cadre de cet ouvrage, le compilateur refuse une telle construction. De ce fait, l'exemple 5.9 est parfaitement représentatif du problème de la portée des déclarations de variables.

Il est fréquent de faire une erreur un peu plus subtile concernant toujours la portée d'une déclaration,

comme l'illustre l'exemple suivant :

Exemple 5.10 :

Voici un programme anodin, refusé par le compilateur :

CompteurFor

```
1 public class CompteurFor {
2   public static void main(String[] args) {
3     for(int i=0;i<5;i++) {
4       System.out.println(i);
5     }
6     System.out.println(i);
7   }
8 }
```

Le compilateur indique l'erreur suivante :

ERREUR DE COMPILATION

```
CompteurFor.java:6: cannot resolve symbol
symbol  : variable i
location: class CompteurFor
    System.out.println(i);
                       ^
1 error
```

La deuxième instruction d'affichage n'est pas acceptée car la variable `i` est considérée comme inaccessible. Si on considère la traduction du `for` en un `while` proposée à la section 5.4.2, on comprend le problème : la déclaration du compteur de la boucle, ici la variable `i` est placée à l'intérieur d'un bloc qui englobe tout le `while` qui traduit le `for`.

Cet exemple permet de déduire la règle suivante : **la portée du compteur d'un `for` se limite au `for` lui-même**. Nous verrons à la section 5.5.3 comment contourner cette limitation afin de pouvoir utiliser le compteur d'un `for` après la boucle.

5.5.2 Initialisation de variables

Comme nous l'avons vu à la section 4.2.3, le compilateur utilise des règles assez strictes pour s'assurer qu'une variable a bien été initialisée avant d'être utilisée. Il vérifie en fait que pour toute exécution envisageable pour le programme, la variable sera initialisée, ce qui pose quelques problèmes avec les `if`s. Nous allons voir que ces problèmes se retrouvent avec les boucles. Commençons par un exemple élémentaire :

Exemple 5.11 :

Voici un programme élémentaire, qui ne fait rien de bien utile, mais illustre simplement le problème :

PasDeValeur

```
1 public class PasDeValeur {
2   public static void main(String[] args) {
3     int j;
4     for(int i=0;i<5;i++) {
5       j=2*i;
6       System.out.println(j);
```

```
7       }
8       int k=2*j;
9       System.out.println(k);
10    }
11  }
```

Le programme est refusé par le compilateur qui affiche le message suivant :

ERREUR DE COMPILATION

```
PasDeValeur.java:8: variable j might not have been initialized
    int k=2*j;
            ^
1 error
```

Cette erreur semble absurde au programmeur qui sait que la boucle `for` s'exécute 5 fois, et qu'à chaque exécution, la ligne 5 donne une valeur à `j`. Comme pour les sélections (`if`), le compilateur ne tient pas compte des conditions : il considère qu'il est éventuellement possible que le `for` ne soit jamais exécuté, et donc que la variable `j` ne soit jamais initialisée.

On peut déduire de cet exemple une règle simple : **le compilateur ne tient pas compte du contenu des boucles `for` et `while` quand il analyse le programme pour savoir si les variables ont bien été initialisées avant d'être utilisées.** Nous sommes dans une situation très similaire à celle de l'exécution conditionnelle, c'est-à-dire du `if` seul.

REMARQUE

Le problème ne se pose pas du tout pour la boucle `do while`. On sait en effet que le corps de cette boucle est exécuté au moins une fois. Si une variable est initialisée dans une boucle `do while`, le compilateur en tiendra compte. On peut donc modifier l'exemple précédent pour obtenir la version suivante, acceptée par le compilateur :

PasDeValeurOk

```
1  public class PasDeValeurOk {
2    public static void main(String[] args) {
3      int j;
4      int i=0;
5      do {
6        j=2*i;
7        System.out.println(j);
8        i++;
9      } while(i<5);
10     int k=2*j;
11     System.out.println(k);
12   }
13 }
```

5.5.3 Forme générale du `for`

Dans la section 5.4.2, nous avons étudié une forme simplifiée de la boucle `for`. La forme générale est un peu plus complexe à maîtriser, mais très pratique.

Syntaxe

La forme générale du `for` est la suivante :

```
for(A;B;C)
  instruction;
```

En général, on préfère remplacer l'instruction par un bloc, ce qui donne :

```
for(A;B;C) {
  instruction 1;
  ...
  instruction n;
}
```

Les trois éléments qui forment la première partie du `for` doivent respecter des règles précises :

- *A* : correspond à la partie d'**initialisation** du `for`. Doit être constitué soit d'une *expression instruction*[6], soit d'une déclaration de variable(s). Peut éventuellement être vide. L'*expression instruction* peut être remplacée par une liste d'*expressions instructions*, séparées par des virgules ;
- *B* : correspond à la "condition" du `for`. Doit être constituée d'une expression de type `boolean`. Peut éventuellement être vide, auquel cas le compilateur considère qu'on a utilisé l'expression `true` ;
- *C* : correspond à la partie de **mise à jour** du `for`. Doit être constitué soit d'une *expression instruction* ou d'une liste d'*expressions instructions*, séparées par des virgules. Peut éventuellement être vide.

Expression instruction

En général, en `Java`, une expression doit être utilisée, par exemple placée dans une variable. Voici un exemple qui illustre cette limitation :

Exemple 5.12 :

Le programme suivant tente de faire apparaître une expression sans l'utiliser :

Expression

```
1 public class Expression {
2   public static void main(String[] args) {
3     2+3.5*3;
4   }
5 }
```

Le compilateur refuse le programme et affiche le message d'erreur suivant :

Erreur de compilation

```
Expression.java:3: not a statement
    2+3.5*3;
       ^
1 error
```

Ce message indique simplement que l'expression n'est pas une instruction (*statement* en anglais).

[6] Traduction assez bancale de *expression statement*.

Une *expression instruction* est une expression `Java` que le compilateur accepte de considérer comme une instruction. Les expressions instructions sont les suivantes :

- les affectations (qui sont donc considérées comme des expressions) ;
- les incrémentations et décrémentations basées sur les opérateurs compacts `++` et `--` ;
- les appels de méthodes (nous avons déjà remarqué ce point à la section 3.3.4 qui indiquait que le résultat d'une méthode ne doit pas obligatoirement être utilisé) ;
- les créations d'objets (nous n'avons pas encore vu ces expressions que nous étudierons à partir du chapitre 7).

Sémantique

La forme générale du `for` est donc relativement complexe, mais fort heureusement, son interprétation reste simple. Considérons en effet la boucle `for` suivante :

```
for(A;B;C) {
  instruction 1;
  ...
  instruction n;
}
```

On peut considérer que le processeur l'exécute comme s'il avait à exécuter la boucle `while` suivante :

```
{
  A;
  while(B) {
    instruction 1;
    ...
    instruction n;
    C;
  }
}
```

En d'autre terme, la sémantique est la suivante :

1. le processeur exécute la (ou les) expression(s) instruction(s) ou la déclaration de variable(s) que constitue *A* ;
2. le processeur évalue l'expression *B* (ou considère la valeur `true` si *B* est vide) ;
3. si l'expression vaut `false`, l'exécution du `for` est terminée ;
4. sinon :
   (a) le processeur exécute les instructions de 1 à *n* ;
   (b) le processeur exécute la (ou les) expression(s) instruction(s) que constitue *C* ;
   (c) le processeur retourne en 2.

Cette sémantique s'illustre de façon assez claire sous forme d'un organigramme, comme le montre la figure 5.5.

REMARQUE

Notons que le `for` est contenu dans un bloc, ce qui signifie que les variables éventuellement déclarées par *A* ne seront utilisables que par le corps de la boucle et par les parties *B* et *C*.

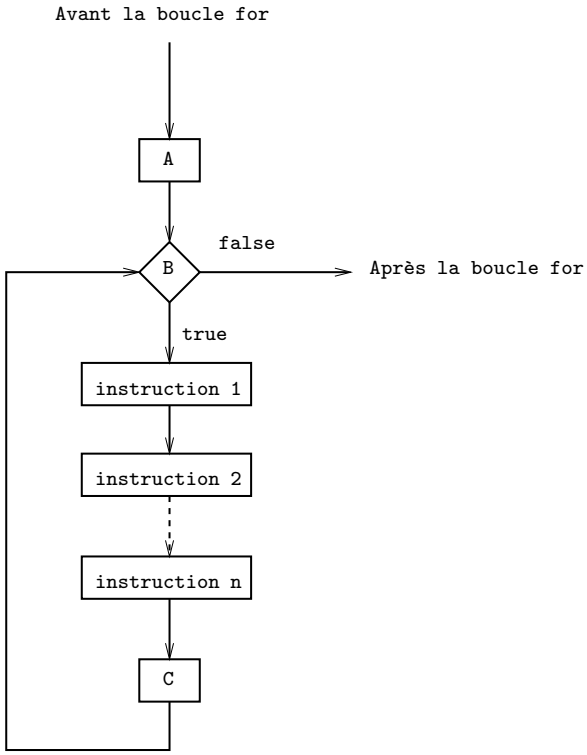

FIG. 5.5 – Sémantique de la forme générale du `for`

Exemples

Voici un exemple simple qui utilise la forme générale du `for` pour pouvoir accéder au compteur de la boucle après la fin de celle-ci :

Exemple 5.13 :

Marche

```
1  import dauphine.util.*;
2  public class Marche {
3    public static void main(String[] args) {
4      Console.start();
5      int j=Console.readInt();
6      int i;
7      for(i=0;i<j;i++) {
8        if(Math.random()>0.5) {
9          j=j-1;
10       } else {
11         j=j+1;
12       }
13     }
14     System.out.println(i);
15   }
16 }
```

Ce programme modélise de façon très simplifiée une sorte de poursuite. Les variables `i` et `j` désignent les positions de deux mobiles. Le mobile `i` va toujours vers la droite (d'où le `i++`).

Au contraire, le mobile `j` se déplace aléatoirement à droite ou à gauche. La boucle s'arrête quand le mobile `i` rattrape ou dépasse le mobile `j`. On affiche alors le nombre de pas effectués par `i`. Comme la variable a été déclarée avant la boucle (ligne 6), elle reste accessible après celle-ci (ligne 14).

La forme générale du `for` permet d'obtenir des comportements très pratiques. On peut par exemple remplacer la mise à jour du compteur de la forme `i++` par une mise à jour plus adapté au problème étudié, comme par exemple `i+=2` qui fait avancer le compteur de 2 en 2. Voici un exemple d'application élémentaire :

Exemple 5.14 :

On souhaite afficher tout les multiples d'un entier donné et strictement inférieur à un autre. Voici une solution très simple :

Multiples

```
1  import dauphine.util.*;
2  public class Multiples {
3    public static void main(String[] args) {
4      Console.start();
5      int n=Console.readInt();
6      int max=Console.readInt();
7      for(int i=n;i<max;i+=n) {
8        System.out.println(i);
9      }
10   }
11 }
```

Grâce au `i+=n`, on passe de `n` en `n` et on affiche ainsi tous les multiples de `n`.

Comme nous l'avons brièvement indiqué à la section 5.4.2, il est parfaitement possible d'utiliser un type réel pour le compteur d'une boucle, comme l'illustre l'exemple suivant :

Exemple 5.15 :

ForReal

```
1  public class ForReal {
2    public static void main(String[] args) {
3      for(double x=0;x<1.5;x+=0.1) {
4        System.out.println(x);
5      }
6    }
7  }
```

Le programme proposé affiche en théorie les réels compris entre 0 et 1.5 en progressant de 0.1 en 0.1. L'affichage produit est pourtant déroutant :

Affichage

```
0.0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
```

```
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
1.3
1.4000000000000001
```

On n'obtient donc pas exactement les valeurs qu'on s'attendait à avoir. Il faut noter que ceci n'a aucun rapport avec l'utilisation d'une boucle `for`, comme le montre le programme suivant :

CalculFaux

```
1 public class CalculFaux {
2   public static void main(String[] args) {
3     double x=0.7;
4     System.out.println(x);
5     x=x+0.1;
6     System.out.println(x);
7   }
8 }
```

Ce programme affiche en effet :

AFFICHAGE

```
0.7
0.7999999999999999
```

Il est donc normal que le calcul donne le même résultat dans le cas de la boucle `for`. En fait, il s'agit d'une illustration éclatante du fait que les types réels sont une approximation des réels au sens mathématique et que les résultats des calculs sont toujours approximatifs (cf à ce sujet la section 2.4.2). Pour l'utilisation dans une boucle, cela peut avoir des conséquences gênantes. Modifions en effet le premier programme pour obtenir la version suivante :

ForRealSansFin

```
1 public class ForRealSansFin {
2   public static void main(String[] args) {
3     for(double x=0;x!=1.5;x+=0.1) {
4       System.out.println(x);
5     }
6   }
7 }
```

Pour le programmeur, cette nouvelle version de la boucle devrait *a priori* fonctionner comme l'ancienne. Il n'en est rien, car les erreurs de calcul font que `x` ne va jamais prendre exactement la valeur `1.5`. De ce fait, le programme ne s'arrête jamais. Le début de son affichage est le suivant :

AFFICHAGE

```
0.0
0.1
```

```
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
1.3
1.4000000000000001
1.5000000000000002
1.6000000000000003
1.7000000000000004
1.8000000000000005
1.9000000000000006
2.0000000000000004
2.1000000000000005
2.2000000000000006
```

Il est donc très important d'être particulièrement attentif à la condition de continuation d'une boucle `for` utilisant un `double` comme compteur.

5.5.4 Boucles emboîtées

Comme une boucle peut répéter le contenu d'un bloc, il est parfaitement possible d'avoir une boucle à l'intérieur d'une autre boucle. On parle alors de *boucles emboîtées*. Voici un exemple d'une telle situation :

Exemple 5.16 :

On considère le programme suivant :

BoucleBoucle

```
1  public class BoucleBoucle {
2    public static void main(String[] args) {
3      for(int i=0;i<3;i++) {
4        System.out.println("i="+i);
5        for(int j=0;j<=i;j++) {
6          System.out.println("j="+j);
7        }
8      }
9    }
10 }
```

L'affichage produit par le programme est le suivant :

Affichage

```
i=0
j=0
```

```
i=1
j=0
j=1
i=2
j=0
j=1
j=2
```

On voit donc (ce qui est parfaitement normal) que la boucle interne est exécutée entièrement pour chaque itération de la boucle externe.

Le seul danger lié aux boucles emboîtées est quelles exercent une grande attraction sur le programmeur débutant qui a tendance à vouloir les utiliser à mauvais escient, comme nous allons le voir dans la suite de cette section.

Les conditions multiples

Il est fréquent dans la pratique d'avoir plus d'une condition à respecter pour continuer à exécuter une boucle. Considérons un exemple simple. On sait que la série $\sum_{i=1}^{n} \frac{1}{i}$ tend vers l'infini quand n tend lui-même vers l'infini. On souhaite étudier expérimentalement la vitesse de divergence de cette suite. Par exemple, étant donné un réel x, on souhaite trouver le plus petit entier n tel que $\sum_{i=1}^{n} \frac{1}{i} > x$. Pour ce faire, on peut penser à une solution simple, comme le propose le programme suivant :

Divergence

```
1  import dauphine.util.*;
2  public class Divergence {
3    public static void main(String[] args) {
4      Console.start();
5      double x=Console.readDouble();
6      double s=0;
7      int i=0;
8      while(s<=x) {
9        i=i+1;
10       s=s+1.0/i;
11     }
12     System.out.println(i);
13     System.out.println(s);
14   }
15 }
```

Mais on sait que $\frac{1}{i}$ devient rapidement très petit et dépasse les capacités de représentation numérique du processeur. Après quelques itérations, la ligne `s=s+1.0/i` n'a plus d'effet : `s` est devenu suffisamment grand et `1.0/i` suffisamment petit pour que le processeur le considère comme nul (comparativement à `s`). Pour éviter que la boucle ne s'arrête jamais, il faut ajouter une condition, par exemple en autorisant au maximum 1000 itérations. On doit donc arrêter la boucle soit si `s` dépasse `x`, soit si `i` dépasse 1000.

Le programme devient le suivant :

DivergenceMax

```
1  import dauphine.util.*;
2  public class DivergenceMax {
```

```
3   public static void main(String[] args) {
4     Console.start();
5     double x=Console.readDouble();
6     double s=0;
7     int i=0;
8     while(s<=x && i<1000) {
9       i=i+1;
10      s=s+1.0/i;
11    }
12    System.out.println(i);
13    System.out.println(s);
14  }
15 }
```

Certains programmeurs débutants pensent **à tort** que les deux conditions doivent se traduire par deux boucles, ce qui donne des **solutions fausses** de la forme suivante :

DivergenceNImporteQuoi

```
1  /* Ce programme ne fait pas ce qu'il est censé faire... */
2  import dauphine.util.*;
3  public class DivergenceNImporteQuoi {
4    public static void main(String[] args) {
5      Console.start();
6      double x=Console.readDouble();
7      double s=0;
8      int i=0;
9      while(s<=x) {
10       for(i=0;i<1000;i++) {
11         i=i+1;
12         s=s+1.0/i;
13       }
14     }
15     System.out.println(i);
16     System.out.println(s);
17   }
18 }
```

Une simple analyse de ce dernier programme montre qu'à la première itération de la boucle `while`, la boucle interne (lignes 9 à 12) calcule les 1000 premiers termes de la suite $\sum_{i=1}^{n} \frac{1}{i}$. Puis, si la condition du `while` n'est pas remplie, la boucle interne recommence le calcul des 1000 premiers termes, mais en les ajoutant au résultat précédent, ce qui donne une suite différente de celle qu'on souhaite étudier... Comme son nom l'indique, ce programme réalise un calcul qui n'a que très peu de lien avec ce que le programmeur souhaitait à l'origine.

Les boucles parallèles

On a parfois besoin de faire évoluer deux compteurs (ou plus généralement deux grandeurs) en parallèle. Considérons l'exemple suivant : étant donnés deux entiers n et p, on souhaite déterminer s'ils se terminent par le même nombre. Plus précisément, on souhaite compter le nombre de chiffres qu'ils possèdent en commun en partant de la gauche et en s'arrêtant à la première différence. Par

exemple, 2357 et 257 possèdent deux chiffres en commun : ils terminent tous les deux par 57. Pour obtenir ce résultat, il suffit d'appliquer la technique proposée à la section 5.2.3 : pour obtenir le dernière chiffre d'un nombre, on calcule le reste de sa division par 10. Pour passer au chiffre suivant, on divise le nombre par 10. Or, ici, on travaille **en parallèle** sur deux nombres, qu'il faudra donc faire évoluer **en même temps**. Pour ce faire on utilise **une seule boucle**, comme l'illustre la solution suivante :

Suffixe

```
1  import dauphine.util.*;
2  public class Suffixe {
3    public static void main(String[] args) {
4      Console.start();
5      int n=Console.readInt();
6      int p=Console.readInt();
7      int nb=0;
8      while(n>0 && p>0 && n%10==p%10) {
9        nb++;
10       n/=10;
11       p/=10;
12     }
13     System.out.println(nb);
14   }
15 }
```

La variable `nb` contient le nombre de chiffres en commun. Toute la subtilité est dans la condition de la boucle `while` (ligne 8). La boucle continue tant que les deux nombres ont encore des chiffres et tant que le chiffre le plus à droite est identique pour les deux nombres. A chaque tour de la boucle, on compte un chiffre en commun de plus et on supprime ce chiffre des deux nombres.

Comme dans le cas des conditions complexes, certains programmeurs débutants pensent **à tort** que l'évolution en parallèle de deux grandeurs doit se traduire par deux boucles emboîtées, ce qui donne des **solutions fausses** de la forme suivante :

SuffixeNImporteQuoi

```
1  /* Ce programme ne fait pas ce qu'il est censé faire... */
2  import dauphine.util.*;
3  public class SuffixeNImporteQuoi {
4    public static void main(String[] args) {
5      Console.start();
6      int n=Console.readInt();
7      int p=Console.readInt();
8      int nb=0;
9      while(n>0) {
10       while(p>0) {
11         p/=10;
12       }
13       if(p%10==n%10) {
14         nb++;
15       }
16       n/=10;
17     }
18     System.out.println(nb);
```

```
19    }
20  }
```

Dans l'exemple proposé, la boucle intérieure à pour effet de placer 0 dans `p`, ce qui n'est pas très utile... Il est très difficile de réaliser une version correcte du programme en utilisant deux boucles emboîtées. De plus, la solution obtenue sera nécessairement moins performante et moins lisible que le programme qui utilise une seule boucle.

Boucles inutiles

Il arrive souvent qu'un calcul qui s'exprime très simplement sous forme de boucles emboîtées puisse s'écrire de façon plus efficace avec une seule boucle. Considérons par exemple le calcul de $\sum_{k=0}^{n} \frac{x^k}{k!}$. On sait que cette série converge vers e^x et on souhaite l'étudier expérimentalement. Or, nous avons vu dans les sections précédentes comment calculer une somme, $k!$ et x^k. Nous pouvons donc proposer la solution suivante :

Exponentielle

```
1  import dauphine.util.*;
2  public class Exponentielle {
3    public static void main(String[] args) {
4      Console.start();
5      int n=Console.readInt();
6      double x=Console.readDouble();
7      double somme=1;
8      for(int k=1;k<=n;k++) {
9        // calcul de x^k
10       double puissance=1;
11       for(int j=1;j<=k;j++) {
12         puissance*=x;
13       }
14       // calcul de k!
15       double fact=1;
16       for(int j=1;j<=k;j++) {
17         fact*=j;
18       }
19       somme+=puissance/fact;
20     }
21     System.out.println(somme+" approche "+Math.exp(x));
22   }
23 }
```

Cette solution est parfaitement correcte, mais elle fait de nombreux calculs inutiles. En effet, posons $u_p = \sum_{k=0}^{p} \frac{x^k}{k!}$, $v_p = x^p$ et $w_p = p!$. On remarque que $u_p = u_{p-1} + \frac{v_p}{w_p}$, $v_p = xv_{p-1}$ et $w_p = pw_{p-1}$. On doit donc calculer **en parallèle** trois suites. On peut même simplifier en définissant $z_p = \frac{x^p}{p!}$ et en remarquant que $z_p = z_{p-1}\frac{x}{p}$. Rien n'oblige à utiliser des boucles emboîtées et en fait, une seule boucle suffit, comme l'illustre la solution suivante :

ExponentielleRapide

```
1  import dauphine.util.*;
2  public class ExponentielleRapide {
3    public static void main(String[] args) {
```

```
4     Console.start();
5     int n=Console.readInt();
6     double x=Console.readDouble();
7     double somme=1;
8     double rapport=1;
9     for(int k=1;k<=n;k++) {
10      rapport*=x/k;
11      somme+=rapport;
12    }
13    System.out.println(somme+" approche "+Math.exp(x));
14  }
15 }
```

Il existe bien entendu des cas pour lesquels une telle simplification n'est pas possible. Si on souhaite calculer $\sum_{k=1}^{n} k^k$ (sans utiliser la méthode `pow` de la classe `Math`), par exemple, on doit utiliser deux boucles emboîtées. En effet, il n'y a pas de relation simple entre k^k et $(k-1)^{k-1}$. De ce fait, à chaque tour de la boucle qui calcule la somme, on doit exécuter une boucle pour calculer k^k. On obtient la solution suivante :

SommePuissance

```
1  import dauphine.util.*;
2  public class SommePuissance {
3    public static void main(String[] args) {
4      Console.start();
5      int n=Console.readInt();
6      long result=0;
7      for(int k=1;k<=n;k++) {
8        long puissance=1;
9        for(int j=1;j<=k;j++) {
10         puissance*=k;
11       }
12       result+=puissance;
13     }
14     System.out.println(result);
15   }
16 }
17
```

Conclusion

Avant d'utiliser des boucles emboîtées, il est important de bien analyser le problème posé. Une mauvaise utilisation de boucles emboîtées peut conduire à un programme incorrect ou à une solution complexe et lente. La bonne approche est la suivante : on commence par essayer de trouver une solution basée sur une boucle unique, puis si le problème semble vraiment impossible à résoudre, on accepte d'ajouter d'autres boucles. Malheureusement, seule l'expérience permet de savoir si une solution donnée est satisfaisante. On ne peut donc pas donner de recette générale...

5.6 L'interruption de boucle

5.6.1 Motivation

Considérons un problème simple : on souhaite déterminer la position de la première occurrence (la première apparition) d'un chiffre donné dans un nombre. On numérote les chiffres en partant de la droite, et en donnant au premier la position zéro[7]. Si on cherche le chiffre 2 dans le nombre 252467, on doit obtenir la position numéro 3, car la première occurrence de 2 dans le nombre en partant de la gauche est le quatrième chiffre. En utilisant toujours les mêmes techniques (quotient et reste de la division par 10), il est facile de réaliser la tâche proposée :

PositionChiffre

```
1  import dauphine.util.*;
2  public class PositionChiffre {
3    public static void main(String[] args) {
4      Console.start();
5      int n=Console.readInt();
6      int nInitial=n;
7      int c=Console.readInt();
8      int pos=0;
9      while(n>0 && n%10!=c) {
10       pos++;
11       n/=10;
12     }
13     if(n>0) {
14       System.out.println("Première occurrence de "+c+" dans "+nInitial
15                          +" en position "+pos);
16     } else {
17       System.out.println(c+" n'apparaît pas dans "+nInitial);
18     }
19   }
20 }
```

Le seule reproche qu'on puisse faire à cette solution est qu'elle n'est pas très facile à analyser à la première lecture. En effet, la condition du `while` mélange deux éléments :

1. `n>0` est une condition qui correspond au parcours du nombre ;
2. `n%10 !=c` est une condition qui correspond à la recherche effectuée.

On mélange donc dans une même condition générale un aspect commun à toutes les manipulations d'entiers chiffre à chiffre (le `n>10`) et quelque chose de particulier à l'application visée, formulé d'ailleurs de façon inverse (on continue à chercher tant que le chiffre courant n'est pas celui qu'on cherche). Pour rendre le programme plus clair, il serait intéressant de pouvoir séparer les deux conditions et de pouvoir exprimer la condition d'arrêt de la recherche de façon directe (en indiquant par exemple qu'on veut arrêter la boucle quand on a trouvé le chiffre recherché). On retrouve un exemple très similaire dans la section 5.5.4 quand on cherche le nombre de chiffres communs à deux nombres.

[7] Par tradition, toutes les collections d'éléments sont numérotées à partir de 0 dans les langages C, C++, `Java`, etc.

5.6.2 L'instruction `break`

Principe

A la section 4.4, et plus particulièrement dans la sous-section 4.4.3, nous avons étudié l'instruction `break` utilisée dans un `switch`. Sa sémantique était alors d'interrompre le `switch`, en passant directement à l'instruction qui le suivait.

Quand on utilise un `break` dans une boucle, on obtient un comportement similaire : quand le processeur rencontre un `break`, il considère que la boucle qu'il est en train d'exécuter est terminée. Il passe donc directement à l'instruction qui suit cette boucle.

Exemples

Commençons par un exemple simple :

Exemple 5.17 :

On considère le programme suivant :

UnBreak

```
1  import dauphine.util.*;
2  public class UnBreak {
3    public static void main(String[] args) {
4      Console.start();
5      int param=Console.readInt();
6      for(int i=1;i<=1000;i++) {
7        System.out.println(i);
8        if(i>=param) {
9          break;
10       }
11     }
12     System.out.println("fin");
13   }
14 }
```

L'interprétation du programme est très simple. Si l'utilisateur saisie une valeur strictement supérieure ou égale à 1000, la boucle s'exécute comme si les lignes 8 à 10 n'existaient pas : elle affiche tout les entiers compris entre 1 et 1000, puis le programme affiche `fin`. Par contre, si `param` est strictement inférieur à 1000, `i` va nécessairement devenir supérieur ou égal à `param`. Dans ce cas, le processeur exécute l'instruction `break`, ce qui provoque l'arrêt de la boucle. Si l'utilisateur saisit la valeur 10, la boucle n'effectue que 10 tours, et affiche donc les entiers de 1 à 10. Quand `i` atteint la valeur 10, le `break` termine la boucle et le programme affiche `fin`. La figure 5.6 donne l'organigramme du programme proposé. On voit clairement que l'instruction `break` provoque une sortie anticipée de la boucle.

Il faut bien noter qu'un `break` termine la boucle dans laquelle il est exécuté, pas une éventuelle boucle englobante, comme l'illustre l'exemple suivant :

Exemple 5.18 :

Considérons le programme suivant :

BreakEmboitee

```
1  import dauphine.util.*;
2  public class BreakEmboitee {
3    public static void main(String[] args) {
```

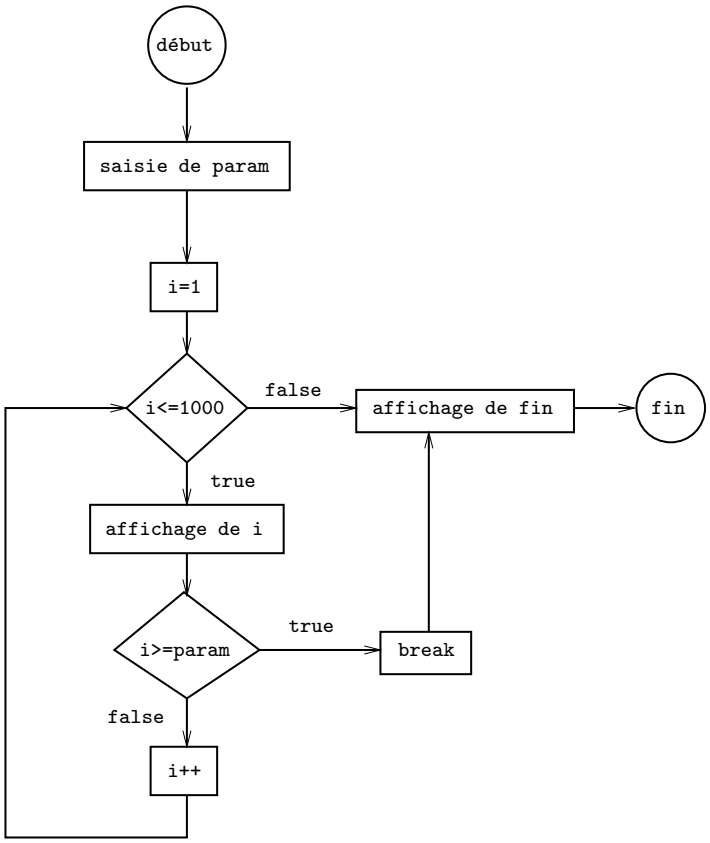

FIG. 5.6 – Organigramme d'un `break` dans une boucle

```
4       Console.start();
5       int param=Console.readInt();
6       for(int i=1;i<=5;i++) {
7         for(int j=1;j<=i;j++) {
8           if (j>param) {
9             break;
10          } else {
11            System.out.println("("+i+","+j+")");
12          }
13        }
14        System.out.println("fin de j");
15      }
16      System.out.println("fin de i");
17    }
18  }
```

Imaginons un instant ce programme sans le `break`. Il affiche alors les couples `(i,j)` pour `i` et `j` compris entre 1 et 5, avec `j`≤`i`. Les affichages des lignes 14 et 16 ne servent qu'à illustrer le comportement des boucles en cas de `break`.

Le comportement du programme avec le `break` dépend de la valeur de `param`, saisie par l'utilisateur. Dès que `j` devient strictement supérieur à `param`, la boucle interne est interrompue. De ce fait, pour `i=3` par exemple, si `param` contient une valeur supérieur à 3, l'exécution de la boucle est complète et affiche donc les couples `(3,1)`, `(3,2)` et `(3,3)`. Par contre, si `param` contient une valeur inférieure strictement à 3, par exemple 2, seuls les couples `(i,j)` pour `j<=param` sont affichés, c'est à dire ici `(3,1)` et `(3,2)`. Puis la boucle interne est arrêtée. Le processeur passe donc à la fin de la boucle externe : `i` est incrémenté de 1 et prend donc la valeur 4. La boucle interne est alors exécutée de nouveau, etc. La figure 5.7 donne l'organigramme du programme étudié.

On voit donc que le programme affiche tous les couples `(i,j)` pour `i` et `j` compris entre 1 et 5, avec `j`≤`i` **et** `j`≤`param`. Si `param` vaut 3, on obtient par exemple l'affichage suivant :

AFFICHAGE

```
(1,1)
fin de j
(2,1)
(2,2)
fin de j
(3,1)
(3,2)
(3,3)
fin de j
(4,1)
(4,2)
(4,3)
fin de j
(5,1)
(5,2)
(5,3)
fin de j
```

```
fin de i
```

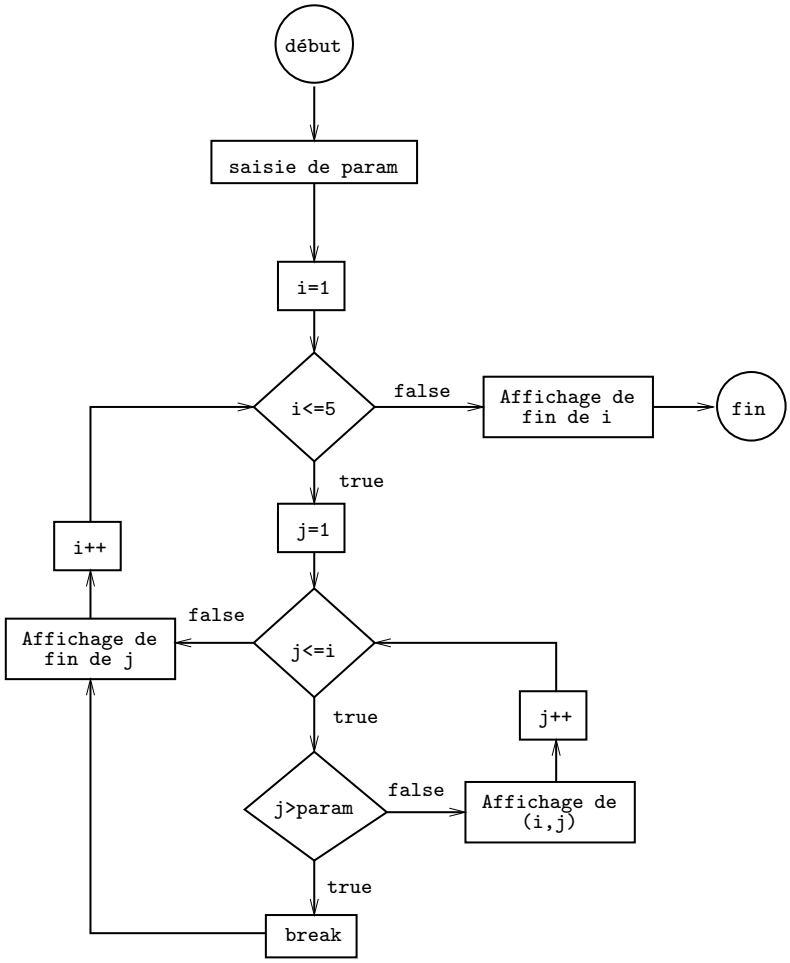

FIG. 5.7 – Organigramme d'un `break` dans des boucles emboîtées

REMARQUE

L'instruction `break` peut s'utiliser avec toutes les boucles, i.e., `while`, `do while` et `for`.

5.6.3 Applications

Commençons par une règle d'or : **il ne faut utiliser l'instruction `break` que si cela peut rendre le programme plus efficace et/ou plus compréhensible.** Reprenons l'exemple cité en introduction (section 5.6.1) :

Exemple 5.19 :

On souhaite donc déterminer la position d'un chiffre dans un nombre. En utilisant un `break`, on rend le programme plus clair car :
– on sépare le parcours des chiffres du nombre de la recherche d'un chiffre particulier ;
– on exprime la condition de sortie de façon directe (c'est-à-dire sans utiliser sa négation).
Voici une solution possible :

PositionChiffreBreak

```
1  import dauphine.util.*;
2  public class PositionChiffreBreak {
3    public static void main(String[] args) {
4      Console.start();
5      int n=Console.readInt();
6      int nInitial=n;
7      int c=Console.readInt();
8      int pos=0;
9      while(n>0) {
10       if(n%10==c) {
11         break;
12       } else {
13         pos++;
14         n/=10;
15       }
16     }
17     if(n>0) {
18       System.out.println("Première occurrence de "+c+" dans "+nInitial
19                          +" en position "+pos);
20     } else {
21       System.out.println(c+" n'apparaît pas dans "+nInitial);
22     }
23   }
24 }
```

On peut même utiliser un `for` (forme étendue) :

PositionChiffreBreakFor

```
1  import dauphine.util.*;
2  public class PositionChiffreBreakFor {
3    public static void main(String[] args) {
4      Console.start();
5      int n=Console.readInt();
6      int c=Console.readInt();
7      int pos=0;
8      int i;
9      for(i=n;i>0;i/=10) {
10       if(i%10==c) {
11         break;
12       } else {
13         pos++;
14       }
15     }
16     if(i>0) {
17       System.out.println("Première occurrence de "+c+" dans "+n
18                          +" en position "+pos);
19     } else {
20       System.out.println(c+" n'apparaît pas dans "+n);
21     }
```

```
22    }
23  }
```

Cette dernière solution est très intéressante, car elle montre comment l'utilisation d'un `for` résume dans une seule ligne l'opération principale réalisée : ici, on étudie tous les chiffres d'un entier. En se servant pleinement de la forme étendue du `for`, qui autorise l'utilisation de plusieurs *expressions instructions*, on obtient une solution que certains programmeurs trouvent encore plus claire :

PositionChiffreBreakFor2

```
1  import dauphine.util.*;
2  public class PositionChiffreBreakFor2 {
3    public static void main(String[] args) {
4      Console.start();
5      int n=Console.readInt();
6      int c=Console.readInt();
7      int i,pos;
8      for(i=n,pos=0;i>0;i/=10,pos++) {
9        if(i%10==c) {
10         break;
11       }
12     }
13     if(i>0) {
14       System.out.println("Première occurrence de "+c+" dans "+n
15                          +" en position "+pos);
16     } else {
17       System.out.println(c+" n'apparaît pas dans "+n);
18     }
19   }
20 }
```

Voici un autre exemple :

Exemple 5.20 :

Reprenons l'exemple de la section 5.5.4 dans lequel on cherche le nombre de chiffres en commun entre deux nombres. Grâce au `break`, on peut donner la formulation suivante :

SuffixeBreak

```
1  import dauphine.util.*;
2  public class SuffixeBreak {
3    public static void main(String[] args) {
4      Console.start();
5      int n=Console.readInt();
6      int p=Console.readInt();
7      int nb=0;
8      while(n>0 && p>0) {
9        if(n%10!=p%10) {
10         break;
11       } else {
12         nb++;
13         n/=10;
14         p/=10;
```

```
15        }
16      }
17      System.out.println(nb);
18    }
19  }
```

Encore une fois, le principal intérêt de cette nouvelle version est de séparer les conditions classiques (`n>0 && p>0`), qui concernent le parcours des chiffres de `n` et `p`, de la condition d'arrêt du comptage (`n%10 !=p %10`).

REMARQUE

Nous ne pouvons pas donner ici beaucoup d'exemples pertinents pour l'application du `break` car nous ne connaissons pas encore les structures adaptées à son emploi. Nous verrons d'autres applications du `break` dans les chapitres qui vont suivre, en particulier dans les chapitres 7 et 8.

5.7 Conseils d'apprentissage

Nous ne saurions trop insister sur l'importance cruciale du présent chapitre : il est **impossible** de programmer sans utiliser de boucles, et, malheureusement, les programmes qui utilisent des boucles (c'est-à-dire les "vrais" programmes) sont beaucoup plus difficile à analyser que les autres. Pour pouvoir continuer son apprentissage, le lecteur devra parfaitement comprendre, puis maîtriser les différentes constructions présentées dans ce chapitre. Comme pour les sélections, la difficulté est avant tout algorithmique, la sémantique des boucles étant pratiquement la même dans tous les langages de programmation. Ce chapitre ne contient donc que très peu d'éléments spécifiques à `Java`. Voici quelques conseils permettant d'organiser l'apprentissage :

- Il faut dans un premier temps comprendre et retenir la **sémantique** des trois boucles étudiées (`do while`, `while` et `for`) : le lecteur doit être capable d'exécuter "sur le papier" un programme utilisant une ou plusieurs boucles, par exemple en représentant l'évolution du contenu des variables pendant l'exécution du programme.
- Dans un deuxième temps, il faut retenir les utilisations typiques de chaque boucle, en s'inspirant des exemples donnés dans le chapitre. On peut retenir les règles très informelles qui suivent :
  - la boucle `for` est adaptée au cas où on connaît *avant* d'exécuter la boucle le nombre d'itérations qu'elle devra effectuer ;
  - la boucle `for` est aussi adaptée au cas où on doit faire varier une valeur entière entre des bornes fixées *avant* d'exécuter la boucle ;
  - la boucle `do while` est adaptée au cas où on doit réaliser une tâche dont on peut déterminer la fin par une condition (par exemple saisir un entier positif, opération qu'on répète tant que l'entier obtenu est négatif ou nul) ;
  - la boucle `do while` est exécutée au moins une fois. La boucle `while` permet de réaliser à peu près les mêmes tâches que la boucle `do while`, mais en évitant cette exécution obligatoire.

  Il faut cependant noter qu'il est toujours possible de réaliser exactement ce que fait une boucle donnée avec une autre boucle. De ce fait, il faut surtout retenir les schémas généraux des utilisations typiques des boucles, comme par exemple celui de la **mise d'un calcul sous forme récurrente**. Il est aussi important de bien comprendre les difficultés liées à l'utilisation de boucles emboîtées.
- La troisième étape de l'apprentissage peut être consacrée à l'étude des éléments plus techniques, comme les problèmes liés à la déclaration et à l'initialisation des variables ou à l'utilisation du `break`.

CHAPITRE 6

Définition de méthodes de classe

Sommaire

Introduction

Dans les chapitres précédents, nous avons découvert les éléments indispensables à l'écriture de programmes. Sans les variables, les sélections et les boucles, il est strictement impossible d'écrire un véritable programme. De plus, même si nous avons donné une présentation spécifique à `Java`, les concepts sous-jacents sont universels. De ce fait, nous avons plutôt étudié l'algorithmique que le langage `Java` lui-même. A partir du présent chapitre, nous allons abordé des éléments d'une autre nature. Nous allons nous intéressé à des constructions plus spécifiques à `Java` et nous allons étudier des concepts qui permettent d'écrire plus efficacement et plus rapidement des programmes, sans pour autant être aussi incontournables que les éléments algorithmes couverts par les premiers chapitres. Il est techniquement possible d'écrire des programmes sans utiliser les techniques que nous allons étudier à partir de maintenant, mais cela ne viendrait à l'idée d'aucun programmeur : l'utilisation de ces techniques permet de gagner du temps et donc de réaliser des programmes plus complexes.

Nous avons vu au chapitre 3 comment utiliser des méthodes déjà définies, par exemple pour faire des calculs scientifiques, pour réaliser un affichage, pour demander à l'utilisateur une valeur, etc. Dans le présent chapitre, nous allons apprendre à écrire nos propres méthodes. Il est en théorie possible d'écrire des programmes avec une méthode unique (la méthode `main`), mais ce n'est vraiment pas une bonne idée. Tout programme comporte en effet des éléments redondants, c'est-à-dire des morceaux de programme quasi-identique. Pour écrire un programme avec des redondances, on peut bien sûr écrire un premier morceau, puis le recopier plusieurs fois en utilisant les possibilités du logiciel d'édition qui permet la saisie du programme. Ce n'est pas une bonne méthode :

1. le programme obtenu est plus gros et donc plus difficile à lire ;

2. si on découvre une erreur dans un des morceaux recopié, il faut la corriger dans tous les autres morceaux, ce qui prend du temps et risque de provoquer des erreurs ;
3. il n'est pas toujours évident pour quelqu'un qui n'a pas écrit le programme que certains morceaux sont presque identiques : le programme est difficile à relire.

Les méthodes constituent une bonne solution au problème : une méthode est une suite d'instructions. Pour écrire un programme qui comporte des redondances, on va tout d'abord écrire chaque morceau à répéter sous forme d'une méthode. Le programme en lui-même sera alors construit sous forme d'une suite d'appels aux différentes méthodes : un morceau répété sera remplacé par plusieurs appels à la même méthode. Comme on ne recopie rien, on évite les problèmes évoqués plus haut.

Nous verrons dans ce chapitre comment écrire une **classe** et une **méthode de classe**. Nous étudierons ensuite le **mécanisme d'exécution d'une méthode**, du point de vue de la méthode appelée, ce qui complétera l'étude effectuée au chapitre 3. Nous entrerons dans les détails en étudiant le **passage de paramètre** puis la **définition d'un résultat**.

Après avoir acquis les éléments techniques, nous passerons à l'aspect méthodologique en donnant des exemples à suivre et des pièges à éviter quand on écrit des méthodes. Nous montrerons enfin comment l'utilisation de **plusieurs classes** permet d'obtenir des programmes mieux organisés et plus rapides à programmer.

6.1 Classes et méthodes

6.1.1 Forme générale d'une classe

Comme nous l'avons vu au chapitre 3 (en particulier à la section 3.1.1), une classe est un groupe d'éléments. Les éléments peuvent être par exemple des méthodes de classes, des constantes de classe, etc. Chaque classe est désignée par un identificateur qui suit les conventions précisées à la section 3.1.4. La forme générale simplifiée d'une classe est la suivante :

```
catégorie class nom de la classe {
  catégorie static élément 1
  catégorie static élément 2
  ...
  catégorie static élément n
}
```

Pour que le compilateur accepte cette définition de classe, il faut qu'elle vérifie certaines contraintes :

- la catégorie de la classe est un concept qui dépasse le cadre de ce cours. Nous utiliserons toujours la catégorie `public`. Ceci aura une conséquence assez importante : le nom du fichier qui contient la définition de la classe devra **obligatoirement** être *nom de la classe*`.java` (cf l'exemple 6.1) ;
- le nom de la classe est un identificateur `Java` et doit donc suivre les règles propres aux identificateurs (cf la section 1.3.1) ;
- certaines définitions d'éléments comprendront un point virgule final obligatoire. Comme pour certains éléments le point virgule est interdit, il faudra être très attentif ;
- nous étudierons peu à peu les catégories des éléments. Dans un premier temps, nous nous contenterons de définir des éléments `public`.

Nous ne donnons ici qu'une forme simplifiée de la définition d'une classe. Dans cette forme, tous les éléments sont des éléments dits **de classe**, à cause du mot clé `static`. Nous verrons au chapitre 9 la forme générale complète d'une classe, ce qui nous permettra de définir des éléments dits **d'instance** (dont la définition ne fera pas apparaître le mot clé `static`).

Exemple 6.1 :

On considère la classe `UneClasse`, qu'on tente de définir dans le fichier `MauvaisNom.java` :

MauvaisNom

```
1 public class UneClasse {
2   public static void main(String[] args) {
3     System.out.println("Exemple");
4   }
5 }
```

Le compilateur refuse le programme et donne le message d'erreur suivant :

ERREUR DE COMPILATION

```
MauvaisNom.java:1: class UneClasse is public, should be declared in a file
named UneClasse.java
public class UneClasse {
       ^
1 error
```

6.1.2 Forme générale d'une méthode de classe

Comme nous l'avons vu au chapitre 3 (en particulier à la section 3.1.1), une méthode est une suite d'instructions, désignée par un identificateur et élément d'une classe. La forme générale d'une méthode de classe est la suivante :

```
catégorie static résultat nom de la méthode (paramètres formels) {
  instruction 1;
  instruction 2;
  ...
  instruction n;
}
```

Pour que le compilateur accepte cette définition de méthode, il faut qu'elle vérifie certaines contraintes :

- comme nous l'avons dit dans la section précédente, nous étudierons progressivement les différentes catégories d'éléments (voir en particulier la section 6.6.4). Pour l'instant, nous nous limiterons à la catégorie `public` ;
- le nom de la méthode est un identificateur `Java` et doit donc suivre les règles propres aux identificateurs ;
- le résultat de la méthode doit être soit un type, soit `void`. Nous étudierons ce point en détail à la section 6.4 ;
- la définition des paramètres formels d'une méthode suit des règles particulières que nous expliciterons à la section 6.3 ;
- nous remarquons qu'une définition de méthode se termine avec une accolade fermante. Dans une définition de classe il sera alors impossible de faire suivre cette accolade par un point virgule ;
- notons pour finir qu'une définition de méthode doit nécessairement être élément d'une définition de classe.

Vocabulaire

Les instructions qui constituent la méthode sont regroupées dans un bloc qu'on appelle le **corps** de la méthode, par opposition à la première ligne (celle qui comporte le nom de la méthode) qu'on appelle **l'en-tête** de la méthode.

6.1.3 Une application `Java`

Les programmes `Java` se répartissent selon plusieurs catégories, comme par exemple les *applets*[1], les *servlets*[2], les applications, etc. Dans ce cours, nous nous intéressons en priorité aux applications. Par définition, une application est un programme `Java` qui peut fonctionner indépendamment de tout autre programme[3], excepté bien sûr la machine virtuelle `Java`.

Comme tout programme `Java`, une application est constituée de plusieurs classes. La seule contrainte que doit satisfaire une application est qu'elle doit comporter au moins une classe (de catégorie `public`) définissant une méthode `main`. Pour exécuter une application, on demande à la machine virtuelle `Java` de lancer une classe de cette application. Le lancement de la classe se traduit par l'exécution de sa méthode `main`. Il est possible dans une même application d'avoir plusieurs classes comportant chacune une méthode `main`. Cela signifie simplement qu'on pourra exécuter l'application de différentes façons, suivant la classe (et donc la méthode `main`) choisie.

Il faut noter que la méthode `main` doit impérativement avoir la forme suivante :

```
public static void main(String[] nom du paramètre) {
  instruction 1;
  instruction 2;
  ...
  instruction n;
}
```

Seul le nom du paramètre de la méthode peut être librement choisi par le programmeur. Tout le reste est fixé. Si le programmeur décide de ne pas respecter la forme précise du `main`, la classe obtenue ne peut pas être exécutée, comme le montre l'exemple suivant :

Exemple 6.2 :

On considère la classe suivante (le sens de l'instruction `return` sera étudié à la section 6.4) :

MauvaisMain

```
1  public class MauvaisMain {
2    public static int main(String[] args) {
3      return 1;
4    }
5  }
```

Ce programme compile sans problème. Par contre, quand on cherche à l'exécuter, on obtient l'erreur suivante :

ERREUR D'EXÉCUTION

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

[1] Ce sont des petits programmes destinés à "améliorer" une page *Web*, en particulier en produisant un affichage et/ou une interactivité impossible à réaliser directement avec le langage HTML (qui décrit les pages *Web*).

[2] Ce sont des programmes qui étendent les possibilités d'un serveur *Web*, par exemple en adaptant dynamiquement la page *Web* proposée par le serveur au navigateur de l'internaute.

[3] Ce n'est le cas ni des *applets* qui fonctionnent à l'intérieur d'un navigateur *Web*, ni des *servlets* qui fonctionnent grâce à un programme particulier appelé *moteur de servlets*.

Cette erreur signifie que la machine virtuelle ne trouve pas la méthode `main` dont elle a besoin pour pouvoir exécuter la classe.

REMARQUE

A la section 1.3, nous avons défini un programme `Java` comme une classe limitée à une méthode `main`. Il s'agissait bien entendu d'une simplification à vocation pédagogique.

6.2 Exécution d'une méthode

Nous avons étudié l'appel de méthode de façon assez exhaustive au chapitre 3. Dans ce chapitre, nous nous étions limité à l'appel de méthodes déjà définies, depuis la méthode `main`. Nous savons donc comment s'interprète un appel de méthode du point de vue de la méthode qui effectue l'appel (la **méthode appelante**). Nous avons voir dans cette section comment s'effectue l'appel du point de vue de la **méthode appelée**.

6.2.1 Exemple

Commençons par étudier un exemple simple :

Exemple 6.3 :

On propose la classe suivante :

AppelMethode

```
1  public class AppelMethode {
2    public static void methode() {
3      System.out.println("Début de la méthode");
4      System.out.println("Fin de la méthode");
5    }
6    public static void main(String[] args) {
7      System.out.println("Début de main");
8      AppelMethode.methode();
9      System.out.println("Fin de main");
10   }
11 }
```

Son exécution produit l'affichage suivant :

AFFICHAGE

```
Début de main
Début de la méthode
Fin de la méthode
Fin de main
```

Ceci correspond à la sémantique décrite dans le chapitre 3 :

1. évaluation des paramètres de la méthode (ici aucun) ;
2. exécution des instructions de la méthode ;
3. utilisation de l'éventuel résultat de la méthode (ici, il n'y en a pas).

6.2.2 Cas général

De façon générale, **toute méthode peut appeler une autre méthode**. Le processeur interprète un appel de la façon suivante :

1. évaluation des paramètres de la méthode appelée ;
2. transmission des valeurs obtenues à la méthode appelée (cf la section 6.3) ;
3. sauvegarde de la position de la tête de lecture dans la méthode appelante ;
4. exécution de la méthode appelée (la tête de lecture est déplacée vers la première instruction de la méthode appelée) ;
5. transmission de l'éventuel résultat à la méthode appelante (cf la section 6.4) ;
6. retour de la tête de lecture à la position sauvegardée dans la méthode appelante et exécution des instructions suivantes.

Nous expliciterons et préciserons les étapes intermédiaires dans la suite du chapitre.

6.2.3 L'appel simplifié

Au chapitre 3, nous avons utilisé systématiquement des méthodes déjà définies, ce qui impose le recours au nom complet de la méthode, c'est-à-dire le nom de la classe, suivi d'un point, suivi du nom de la méthode.

Quand une classe comporte plusieurs méthodes, il est fréquent que certaines d'entre elles appellent d'autres méthodes de la même classe. On peut alors utiliser un appel simplifié : il suffit de donner le nom de la méthode, sans préciser le nom de la classe. Le nom de la méthode seule est appelé son **nom simple** ou encore son **nom relatif**. Voici un exemple d'utilisation du nom simple :

Exemple 6.4 :

On considère le programme suivant :

NomSimple

```
1  public class NomSimple {
2    public static void a() {
3      System.out.println("méthode a");
4    }
5    public static void b() {
6      a();
7      System.out.println("méthode b");
8    }
9    public static void main(String[] args) {
10     a();
11     b();
12   }
13 }
```

Ce programme affiche :

Affichage

```
méthode a
méthode a
méthode b
```

Le programme illustre ainsi l'utilisation du nom simple et montre qu'une méthode autre que `main` peut appeler une autre méthode.

6.2.4 Absence d'ordre

Il est important de noter qu'une classe n'est pas un ensemble ordonné. Si une méthode appelle une autre méthode de la même classe, la méthode appelée ne doit pas obligatoirement être définie avant la méthode appelante dans le texte de la classe. Voici un exemple simple illustrant cette règle :

Exemple 6.5 :

On considère la classe suivante :

PasDOrdre

```
1  public class PasDOrdre {
2    public static void main(String[] args) {
3      première();
4    }
5    public static void première() {
6      System.out.println("première");
7      deuxième();
8    }
9    public static void deuxième() {
10     System.out.println("deuxième");
11   }
12 }
```

Quand elle est exécutée, elle produit l'affichage suivant :

AFFICHAGE

```
première
deuxième
```

REMARQUE

En général, on s'arrange pour définir une méthode avant qu'elle soit utilisée, car cela permet d'obtenir un programme plus facile à lire.

6.2.5 Les variables sont locales

Comme nous l'avons vu à la section 4.2.2, chaque variable possède une portée, qui est limitée aux instructions qui suivent sa déclaration et qui appartiennent au même bloc que celle-ci. Or, une méthode est délimitée par un bloc et toute variable déclarée dans une méthode aura pour portée les instructions **de la méthode** qui suivent la déclaration. Ceci a une conséquence simple mais importante : les variables déclarées dans une méthode ne sont pas utilisables en dehors de celle-ci. On dit que les variables sont **locales** à la méthode[4].

[4] Il est possible de déclarer des variables globales (c'est-à-dire accessibles par plusieurs méthodes), mais c'est en général une assez mauvaise idée. Nous ne développerons pas ce point pour l'instant.

L'exemple suivant illustre ce point :

Exemple 6.6 :

On considère le programme suivant :

VariablesLocales

```
1  public class VariablesLocales {
2    public static void a() {
3      int x=2;
4      System.out.println(x);
5    }
6    public static void b() {
7      x=3; // tentative d'accès à la variable déclarée dans a
8      System.out.println(x); // seconde tentative d'accès
9    }
10   public static void main(String[] args) {
11     b();
12   }
13 }
```

Comme la méthode `b` tente d'accéder à une variable de la méthode `a`, le compilateur refuse le programme et donne le message d'erreur suivant :

Erreur de compilation

```
VariablesLocales.java:7: cannot resolve symbol
symbol  : variable x
location: class VariablesLocales
    x=3; // tentative d'accès à la variable déclarée dans a
    ^
VariablesLocales.java:8: cannot resolve symbol
symbol  : variable x
location: class VariablesLocales
    System.out.println(x); // seconde tentative d'accès
                       ^
2 errors
```

On remarque que le compilateur ne fait aucune allusion à la méthode `a`. En effet, il ne peut pas savoir que le `x` était une tentative d'utilisation d'une variable de la méthode `a`. Pour le compilateur, on tente simplement d'utiliser un symbole qui n'a pas été défini.

6.2.6 Appel et variables locales

Il faut bien noter que lors d'un appel de méthode, le passage dans la méthode appelée rend en quelque sorte "invisibles" les variables de la méthode appelante. Plus précisément, **les valeurs des variables de la méthode appelantes ne peuvent pas être modifiées par les instructions de la méthode appelée (et vice-versa)**. Voici un exemple qui illustre ce phénomène :

Exemple 6.7 :

On considère la classe suivante :

VariablesIndependantes

```
1  public class VariablesIndependantes {
2    public static void a() {
```

```
3       int x=2;
4       System.out.println("a : "+x);
5     }
6     public static void b() {
7       int x=3;
8       a();
9       System.out.println("b : "+x);
10    }
11    public static void main(String[] args) {
12      int x=4;
13      a();
14      b();
15      System.out.println("main : "+x);
16    }
17  }
```

Notons tout d'abord que la déclaration de `x` dans les trois méthodes est indispensable. Si une déclaration manque, le programme ne peut pas être compilé. L'affichage produit est le suivant :

AFFICHAGE

```
a : 2
a : 2
b : 3
main : 4
```

Ceci montre que chaque méthode possède sa propre version de la variable `x` et que la modification de son contenu n'a aucune influence sur les variables des autres méthodes.

Pour bien comprendre le fonctionnement du programme lors d'un appel de méthode, on peut d'utiliser une représentation graphique de la mémoire, comme celle proposée à la section 2.1.4. La figure 6.1 donne l'état de la mémoire après l'exécution de la ligne 12 de la méthode `main` du programme proposé dans l'exemple précédent. Cette ligne a pour effet de créer la variable `x` et de lui donner la valeur 4.

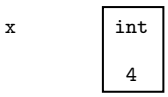

FIG. 6.1 – méthode `main`, lignes 12, 13 (après l'appel), 14 (après l'appel) et 15

Quand la ligne 13 est exécutée, le processeur quitte la méthode `main` pour passer dans la méthode `a`. Avant de quitter la méthode `main`, le processeur effectue une manipulation de la mémoire qui va isoler les variables de la méthode appelante des éventuelles variables de la méthode appelée. Sur la figure 6.2, cette manipulation est représentée par une ligne en tirets. Cette ligne symbolise une **barrière** entre les deux zones mémoires. Quand la ligne 3 de la méthode `a` est exécutée, une variable `x` est créée dans la zone réservée à la méthode. Ceci explique que la figure 6.2 indique deux variables `x`, chacune avec une valeur différente.

Quand une méthode se termine, le bloc qui correspond à son corps est lui aussi terminé. Ceci entraîne la destruction dans la mémoire de toutes les variables déclarées dans ce bloc. Quand le processeur quitte la méthode `a`, il détruit donc la variable `x` déclarée dans cette méthode. Il retourne

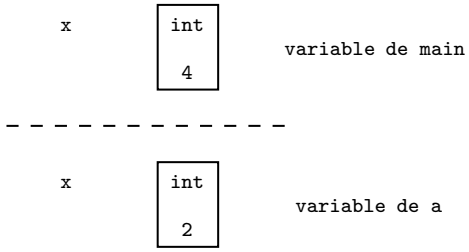

FIG. 6.2 – méthode `a`, ligne 3 (méthode appelante `main`, ligne 13)

ensuite dans la méthode appelante et supprime la barrière de séparation avec la méthode appelée. De ce fait, la mémoire revient ici dans l'état représenté par la 6.1.

L'appel à la méthode `b` de la ligne 14 provoque un comportement similaire à l'appel de la ligne 13. La barrière de séparation entre les zones de la mémoire se met en place, ce qui conduit à la structure de la mémoire représentée par la figure 6.3.

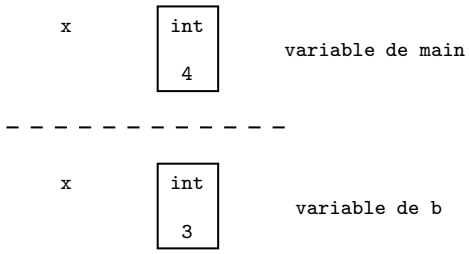

FIG. 6.3 – méthode `b`, lignes 7, 8 (après l'appel) et 9 (méthode appelante `main`, ligne 14)

La ligne 8 de la méthode `b` est un appel de la méthode `a`. Une fois de plus, le processeur doit créer une barrière pour isoler la mémoire de la méthode appelante de celle de la méthode appelée. La structure de la mémoire est de nouveau modifiée et on obtient le résultat représenté par la figure 6.4.

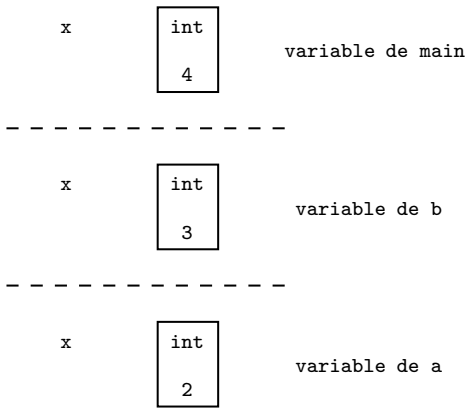

FIG. 6.4 – méthode `a`, ligne 3 (méthode appelante `b`, ligne 8)

Quand l'appel de `a` est terminé, le processeur revient dans la méthode `b`, après avoir supprimé les variables de `a`. La barrière n'existe plus et la mémoire revient dans l'état décrit par la figure 6.3. Quand la méthode `b` se termine à son tour, ses variables sont détruites et le processeur revient dans la méthode `main`. La mémoire retrouve finalement l'état représenté par la figure 6.1.

REMARQUE

La représentation par une barrière de la séparation entre différentes zones mémoires est simplement une visualisation pratique des conséquences du mécanisme de gestion de la mémoire par `Java`. La connaissance exacte des techniques employées par la machine virtuelle est totalement inutile pour la compréhension des programmes, c'est pourquoi nous nous contenterons de cette représentation simplifiée. Le lecteur intéressé par les détails techniques pourra se reporter à [8], plus particulièrement à la section 3.6 sur les *Frames*.

6.3 Passage de paramètres

6.3.1 Motivation

Les sections précédentes montrent qu'il est impossible d'utiliser des variables pour faire communiquer deux méthodes. Or, dans la pratique, une telle communication est indispensable pour obtenir des méthodes intéressantes. La communication *depuis* la méthode *appelante vers* la méthode *appelée* se fait par l'intermédiaire des **paramètres** de la méthode appelée. Pour simplifier, on peut dire que les paramètres d'une méthode jouent le rôle des données d'un algorithme. A partir de ces informations, la méthode va réaliser un certain traitement, comme par exemple afficher à l'écran un résultat.

REMARQUE

Comme nous l'expliquerons dans la section 6.5.3, il est préférable que les méthodes définissent un résultat plutôt que de l'afficher. Comme nous étudierons la définition d'un résultat à la section 6.4, nous nous contenterons dans la présente section de donner des exemples de méthode qui effectuent des affichages. Cette situation est provisoire et ne doit pas laisser croire qu'il est intéressant d'écrire de telles méthodes.

6.3.2 Déclaration des paramètres d'une méthode

Syntaxe

Comme nous l'avons vu à la section 6.1.2, une déclaration de méthode de classe comporte toujours la déclaration des **paramètres formels** de cette méthode. Voici la forme générale de cette déclaration :

type 1 identificateur 1,...,*type n identificateur n*

Chaque identificateur définit un **paramètre formel** de la méthode, dont le type est donné par le mot clé qui le précède. Une méthode de classe a donc la forme générale suivante :

```
catégorie static résultat nom de la méthode(type 1 identificateur 1,
                                            ...,type n identificateur n) {
  instruction 1;
  instruction 2;
  ...
  instruction n;
}
```

Il est important de noter qu'une méthode peut parfaitement avoir une déclaration de paramètre vide, comme l'ont montré les exemples étudiés jusqu'à présent. Notons d'autre part que l'utilisation

de la notation `...` est, comme partout ailleurs dans cet ouvrage, une facilité d'écriture indiquant qu'un nombre arbitraire de paramètres formels peut être utilisé.

Exemple 6.8 :

Voici un exemple simple de méthode[5] déclarant des paramètres formels :

```
public static void a(int x,int y,double z) {
  System.out.println(x+" "+y+" "+z);
}
```

Contraintes

La déclaration des paramètres formels d'une méthode ressemble à une déclaration de variables. Il ne faut cependant pas les confondre :

- si plusieurs paramètres possèdent le même type, on doit quand même répéter le type pour chaque paramètre (comme l'illustre l'exemple précédent) ;
- chaque paramètre est séparé du précédent par une virgule, même si les types des paramètres diffèrent ;
- on ne peut pas donner de valeur initiale à un paramètre dans la déclaration.

Signature

A la section 3.3.2, nous avons défini la notion de signature d'une méthode. Cette signature est déterminée par la déclaration des paramètres formels de la méthode. Pour obtenir la signature d'une méthode, il suffit de supprimer dans sa déclaration les identificateurs des paramètres formels. Si on reprend l'exemple 6.8, la méthode `a` qui a été définie possède donc la signature `a(int,int,double)`. Cela signifie que pour appeler la méthode `a`, il faut fournir (dans cet ordre) deux valeurs d'un type moins général que `int` et une valeur d'un type moins général que `double` (revoir le chapitre 3 au sujet du typage des appels de méthodes).

Quand on écrit la déclaration des paramètres formels d'une méthode, on définit donc la signature de celle-ci. De ce fait, on indique **de quelle façon la méthode pourra être utilisée**, en accord avec les règles de typage des appels de méthodes. Pour résumer et simplifier, les paramètres formels indiquent les valeurs attendues lors de l'appel : pour la méthode donnée en exemple, on attend ainsi deux `int`s et un `double`, dans cet ordre.

Variables pour la méthode

Pour la méthode qui déclare des paramètres formels, ceux-ci jouent le rôle d'autant de variables. De plus, chaque variable est considérée comme initialisée avant d'exécuter la méthode. Ceci explique pourquoi la méthode `a`, proposée dans l'exemple 6.8, peut utiliser le contenu de `x`, `y` et `z` sans avoir ni déclaré ni initialisé ces variables.

La valeur initiale des paramètres formels est déterminée par le mécanisme du passage de paramètres décrit dans la section suivante. Notons que les variables correspondant aux paramètres formels d'une méthode ne sont utilisable que dans le corps de celle-ci.

6.3.3 Mécanisme du passage de paramètres

Il nous reste maintenant à expliquer la façon dont le processeur interprète l'appel d'une méthode quand celle-ci demande des paramètres.

[5] Nous donnons ici le code de la méthode sans le placer dans une classe. Nous utiliserons régulièrement cette simplification pour éviter d'alourdir l'ouvrage.

Exemple introductif

Reprenons l'exemple de la section précédente :

Exemple 6.9 :

On considère le programme suivant :

Passage

```
1  public class Passage {
2    public static void a(int x,int y,double z) {
3      System.out.println(x+" "+y+" "+z);
4    }
5    public static void main(String[] args) {
6      a(2,3,5.0);
7      int u=3;
8      a(u,2*u,7.0);
9    }
10 }
```

Ce programme affiche :

Affichage

```
2 3 5.0
3 6 7.0
```

On devine que dans la méthode `a`, les paramètres formels se comportent comme des variables dont les valeurs initiales sont données par les paramètres effectifs, c'est-à-dire les valeurs utilisées lors de l'appel.

L'exécution de l'appel

Nous avons revu à la section 6.2.2 les différentes actions effectuées par le processeur pour exécuter un appel de méthode. Revenons sur ces actions quand la méthode déclare des paramètres formels :

1. le processeur commence par évaluer les paramètres de l'appel. Ce sont les **paramètres effectifs** de la méthode appelée ;
2. le processeur transmet à la méthode appelée les valeurs obtenues, grâce au mécanisme suivant[6] :
   (a) le processeur considère la déclaration de paramètres formels comme une déclaration de variables et crée donc, **dans la zone mémoire de la méthode appelante**, autant de variables qu'il y a de paramètres formels ;
   (b) le processeur place la **valeur du paramètre effectif** dans la variable représentant le paramètre formel correspondant. Ceci est possible car le compilateur a vérifié que la signature de l'appel était compatible avec la signature de la méthode (cf la section 3.3.2) ;

[6] Nous présentons ici un mécanisme simplifié qui décrit relativement fidèlement le *résultat* de la transmission des paramètres. Le lecteur intéressé par une description exacte du *véritable* mécanisme utilisé se reportera à [8], en particulier à la section 3.6.

3. le processeur sauvegarde la position de la tête de lecture dans la méthode appelante. Il place à ce moment la barrière (décrite à la section 6.2.6) entre la zone mémoire de la méthode appelante et celle de la méthode appelée. **Les variables correspondant aux paramètres formels sont placées dans la zone mémoire de la méthode appelée**[7] ;
4. le processeur exécute la méthode appelée. Quand l'exécution est terminée, toutes les variables de la zone mémoire de la méthode appelée sont **supprimées**, ce qui **inclut les variables correspondant aux paramètres formels** ;
5. l'éventuel résultat de la méthode appelée est transmis à la méthode appelante (cf la section 6.4) ;
6. le processeur continue l'exécution de la méthode appelante.

Exemple 6.10 :

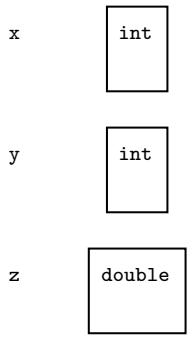

FIG. 6.5 – Création des paramètres formels de la méthode `a`

Reprenons l'exemple 6.9 et analysons l'appel de la ligne 6. Comme les paramètres effectifs de cet appel sont des valeurs littérales, l'évaluation (étape 1) est immédiate. La méthode appelée déclare trois paramètres formels. Le processeur fabrique donc trois variables (deux de type `int`s et une de type `double`), ce qui place la mémoire dans l'état représenté par la figure 6.5. Quand les variables sont crées, le processeur y place les valeurs des paramètres effectifs, ce qui donne le nouvel état de la mémoire indiqué par la figure 6.6.

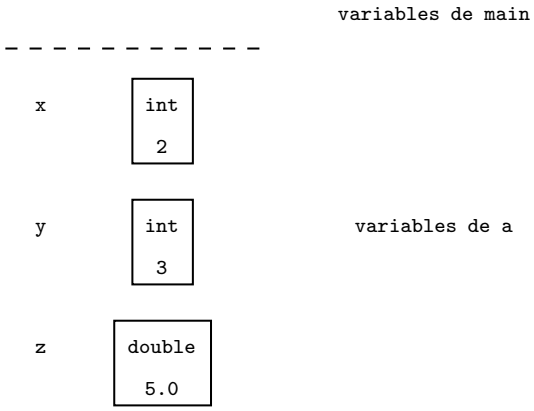

FIG. 6.6 – Recopie des paramètres effectifs dans les paramètres formels

[7] C'est à ce moment précis qu'il y a transmission d'information : les variables sont crées dans la zone mémoire de la méthode appelante, puis la barrière est placée de telle sorte que tout se passe comme si les variables avaient en fait été crées dans la zone mémoire de la méthode appelée.

Le processeur place ensuite la barrière de séparation entre les zones mémoires. La zone de la méthode appelante (ici le `main`) reste vide car cette méthode n'a déclaré aucune variable[8]. la figure 6.6 indique les deux zones et la barrière de séparation.

Grâce au mécanisme de transmission la méthode `a` dispose maintenant de trois variables `x`, `y` et `z` dont les valeurs initiales sont celles des paramètres effectifs de l'appel. Il y a bien transfert d'information de puis la méthode appelante (`main`) vers la méthode appelée (`a`). On comprend ainsi pourquoi le premier appel produit l'affichage `2 3 5.0`.

6.3.4 Récapitulation

Pour transmettre des informations depuis la méthode appelante vers la méthode appelée, on utilise donc le mécanisme du passage de paramètres. La méthode appelée déclare des **paramètres formels** qui sont interprétés de la façon suivante :

- pour le compilateur, les paramètres formels précisent la signature de la méthode : ils indiquent donc le nombre de valeurs attendues par la méthode lors d'un appel, ainsi que le type de chaque valeur (avec un ordre précis) ;
- pour le compilateur, les paramètres formels correspondent à des variables locales de la méthode appelée, déjà initialisées. De ce fait, on peut utiliser directement le contenu des paramètres formels dans la méthode appelée ;
- pour le processeur, les paramètres formels sont d'abord créés comme des variables locales de la méthode appelante. Ils sont initialisées avec les valeurs utilisées lors de l'appel (les **paramètres effectifs**), puis placés dans la zone mémoire de la méthode appelée, ce qui réalise la communication effective entre les deux méthodes.

REMARQUE

L'information est transmise sous forme de valeurs. Techniquement on dit que le passage de paramètre utilisé par `Java` est un **passage par valeur**.

6.3.5 Subtilités du passage de paramètres

La sémantique du passage de paramètre est suffisamment complexe pour qu'il soit difficile de se faire une idée précise de ses conséquences. Dans cette section, nous présentons quelques difficultés liées au passage de paramètres, qui sont essentiellement des conséquences du passage par valeur.

Comme nous l'avons vu à la section 6.2.6, le contenu des variables d'une méthode ne peut pas être modifié par des opérations réalisées dans une autre méthode. Or, l'utilisation du passage de paramètre, même s'il ne remet pas en cause ce principe, peut entraîner des confusions, comme le montre l'exemple suivant :

Exemple 6.11 :

On considère la classe suivante :

PasDeModification

```
1  public class PasDeModification {
2    public static void a(int x) {
3      x=1;
4      System.out.println(x);
5    }
6    public static void main(String[] args) {
7      int x=-2;
```

[8] En fait, comme la méthode `main` possède un paramètre formel (`args`), on devrait en toute rigueur le représenter dans la mémoire.

```
8       a(x);
9       System.out.println(x);
10    }
11  }
```

On pourrait croire que les variables `x` des méthodes `a` et `main` sont les mêmes, ou tout au moins qu'elles sont liées, c'est-à-dire qu'une modification effectuée dans la méthode `a` se répercute dans la méthode `main`. Or, il n'en est rien, et l'affichage obtenu est le suivant :

AFFICHAGE

```
1
-2
```

Pour bien comprendre ce qui se passe, il suffit d'analyser l'appel de la ligne 8 :

- comme pour tout appel de méthode, le processeur commence par évaluer le paramètre effectif de l'appel, c'est-à-dire ici le contenu de la variable `x`. Pour la suite de l'exécution, tout se passe comme si l'appel était `a(-2)`. Ceci constitue le point crucial à retenir : pour le processeur, les paramètres effectifs d'un appel sont toujours des **valeurs**. De ce fait, lors de l'exécution de la méthode `a`, le processeur aura "oublié" que l'appel du programme était `a(x)`. Il ne pourra donc pas faire lien entre la variable `x` de la méthode `main` et le paramètre formel `x` de la méthode `a` ;
- tout le processus de passage de paramètre est ensuite effectué par le processeur. En résumé, on se retrouve avec dans la mémoire deux variables `x`, celle du `main` et celle qui est destinée à recevoir la valeur du paramètre effectif de la méthode `a`. Juste avant l'appel proprement dit, les deux variables `x` ont la même valeur, à savoir `-2` ;
- quand le processeur pose la barrière, les deux variables deviennent complètement séparées (cf la figure 6.7). Il est maintenant impossible à la méthode `a` de modifier la variable `x` du `main` (et vice-versa) ;

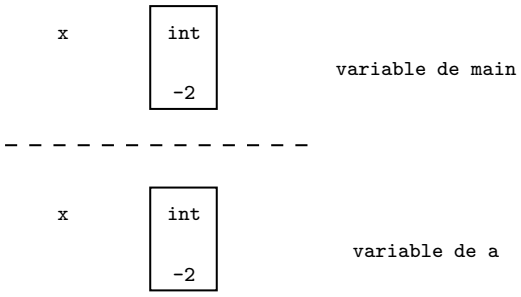

FIG. 6.7 – méthode `main`, ligne 8

- dans la méthode `a`, le processeur modifie la valeur de la variable correspondant à son paramètre formel (ligne 3). Cette variable n'a **aucun lien direct** avec la variable `x` de la méthode `main`. Après la ligne 3, le `x` de `a` contient la valeur `1`, alors que le `x` du `main` est bien entendu inchangé et contient donc toujours `-2` (voir la figure 6.8) ;
- le processeur exécute alors la ligne 4, qui produit l'affichage `1`, puis il retourne dans la méthode `main`, après avoir détruit la variable de la méthode `a`. On se retrouve donc avec une seule variable `x`, qui n'a pas été modifiée (voir la figure 6.9). La dernière instruction du programme (ligne 9) produit donc l'affichage de la valeur `-2`.

On voit donc que l'utilisation du même nom pour un paramètre formel et pour une variable peut induire en erreur. C'est pourquoi il est important de bien connaître le mécanisme de passage des

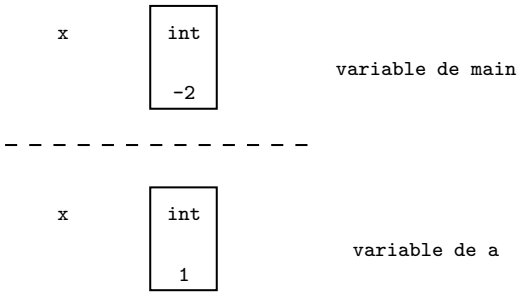

FIG. 6.8 – méthode `a`, après la ligne 3

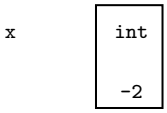

FIG. 6.9 – méthode `main`, après la ligne 8

paramètres et surtout de garder à l'esprit la règle énoncée à la section 6.2.6 : **les valeurs des variables d'une méthode ne peuvent pas être modifiées par les instructions d'une autre méthode**.

REMARQUE

Dans certains langages (par exemple le `C` et le `C++`), cette règle est fausse, ce qui rend parfois les programmes plus difficiles à comprendre[9]. Pour employer des termes techniques, `Java` n'utilise que le passage par valeur. Dans l'exemple précédent, l'appel `a(x)` est en fait interprété comme l'appel `a(-2)`. De ce fait, c'est une *valeur* (ici numérique) qui est transmise à la méthode appelée. Dans d'autres langages, on autorise la transmission de la *variable* elle-même : la variable `x` de la méthode `a` est alors la même que celle de la méthode `main`. Cela complique bien sûr énormément l'analyse des programmes.

Une autre conséquence du passage par valeur est illustrée par l'exemple suivant :

Exemple 6.12 :

On considère le programme suivant :

Ordre

```
1  public class Ordre {
2    public static void a(int x,int y) {
3      System.out.println(x+" "+y);
4    }
5    public static void main(String[] args) {
6      int x=-2;
7      int y=5;
8      a(y,x);
9    }
10 }
```

[9] La contrepartie est qu'on peut parfois écrire des programmes plus efficaces. L'adoption de règles simples se traduit parfois en programmation par une perte d'efficacité. C'est le prix à payer pour pouvoir écrire des programmes plus faciles à relire et donc plus faciles à maintenir.

Ce programme affiche :

AFFICHAGE

```
5 -2
```

Une analyse naïve du programme pourrait laisser croire qu'il doit afficher `-2 5`. Or, le processeur ne fait aucun lien entre les variables `x` et `y` de la méthode `main` et les paramètres formels du même nom de la méthode `a`. De plus, pour exécuter l'appel de la ligne `8`, le processeur commence par évaluer les paramètres effectifs. De ce fait, tout se passe comme si on effectuait l'appel `a(5,-2)`, ce qui explique l'affichage obtenu.

Il faut donc être très attentif quand on utilise une méthode. Pour le processeur, l'association entre un paramètre effectif et le paramètre formel correspondant se fait exclusivement en fonction de la **position** dans la liste des paramètres. L'exemple présenté est bien sûr pathologique, mais il a le mérite d'illustrer de façon radicale le mécanisme de passage par valeur.

6.4 Résultat d'une méthode

6.4.1 Motivation

Nous avons vu dans la section précédente que l'utilisation de paramètres permet de transmettre des valeurs depuis la méthode *appelante* vers la méthode *appelée*. Les limitations du passage par valeur n'autorisent pas de transmission dans l'autre sens. Pour l'instant, nous ne savons pas comment transmettre une valeur depuis la méthode *appelée* vers la méthode *appelante*. Or cela est particulièrement pratique. Considérons par exemple la méthode `sin` de la classe `Math`. Le seul moyen d'utiliser cette méthode est de prendre en compte son *résultat*. Or, ce résultat est une valeur numérique (ici de type `double`) qui est calculée par la méthode à partir du paramètre effectif de l'appel. Il y a donc bien transmission de la méthode appelée vers la méthode appelante. La seule autre solution serait d'afficher le résultat, par exemple par un `System.out.println`, solution que nous avons utilisée dans les exemples précédents. L'affichage n'est pas très utile en général car il transmet une information *à l'utilisateur* du programme, pas aux instructions de celui-ci. Quand une méthode affiche son résultat, la méthode appelante ne peut pas l'utiliser et, en général, la méthode appelée est donc relativement inutile (voir la section 6.5.3 pour un développement sur ce sujet).

Dans cette section, nous allons voir comment définir le **résultat** d'une méthode, afin de proposer une communication complète entre méthodes.

6.4.2 Exemple

Commençons par un exemple simple :

Exemple 6.13 :

On considère le programme suivant :

Resultat

```
1  public class Resultat {
2    public static int f(int x) {
3      return 2*x+1;
4    }
5    public static void main(String[] args) {
6      int u=f(2);
7      System.out.println(u);
8      System.out.println(2*u);
```

```
9       u=f(3);
10      System.out.println(u);
11    }
12  }
```

Ce programme affiche :

AFFICHAGE

```
5
10
7
```

Il est relativement clair que l'instruction `return` de la ligne 3 définit le résultat de la méthode `f` comme étant l'expression qui suit le `return`.

Avant d'entrer dans les détails, on peut faire une comparaison avec une méthode sans résultat :

Exemple 6.14 :

On considère le programme suivant :

PasDeResultat

```
1  public class PasDeResultat {
2    public static void f(int x) {
3      System.out.println(2*x+1);
4    }
5    public static void main(String[] args) {
6      f(2);
7      // aucun moyen d'afficher le résultat de f multiplié par 2
8      f(3);
9    }
10 }
```

Ce programme affiche :

AFFICHAGE

```
5
7
```

Il est strictement impossible de réaliser des opérations sur le résultat de la méthode `f`, tout simplement parce qu'elle n'en définit pas (au sens de `Java`). Le seul moyen d'afficher le résultat et le double du résultat est de modifier la méthode `f`. Mais dans ce cas, il devient impossible d'obtenir **dans le même programme** à la fois l'affichage d'une valeur pour un paramètre effectif donné (l'appel `f(3)`) et de deux valeurs pour une autre valeur pour le paramètre effectif (l'appel `f(2)`). La méthode `f` est donc ici singulièrement limitée.

6.4.3 Déclaration du résultat d'une méthode

Principe et syntaxe

Quand on définit une méthode, on peut indiquer qu'elle propose un résultat. Il est très important de noter qu'une méthode doit respecter la règle suivante : ou bien elle ne définit **jamais** de résultat, ou bien elle définit **toujours** exactement **un** résultat. Notons bien qu'on parle ici de la notion de

résultat définie au chapitre 3 (en particulier à la section 3.3.3). Il s'agit donc pour une méthode de proposer une **valeur** qui remplacera l'appel de la méthode après l'exécution de celui-ci.

Nous avons déjà étudié la syntaxe générale d'une déclaration de méthode, en particulier aux sections 6.1.2 et 6.3. Rappelons que juste après le mot clé `static`, on doit indiquer le résultat de la méthode. On peut choisir d'indiquer :

- soit `void` : ce mot clé indique que la méthode ne définit aucun résultat ;
- soit un **type quelconque** qui précise le type du résultat de la méthode.

Conséquence pour l'utilisation

Tout appel de méthode est considéré comme une expression, plus précisément comme une expression instruction (voir la section 5.5.3). Nous savons que le résultat d'une méthode ne doit pas obligatoirement être utilisé (c'est justement pour cette raison qu'on parle d'expression instruction), cf la section 3.3.4. Selon la déclaration de la méthode, certaines contraintes d'utilisation de celle-ci apparaissent :

- si la méthode ne possède pas de résultat (`void`), alors le seul moyen de l'utiliser est de faire un appel seul, comme nous l'avons déjà vu à la section 3.3.4 ;
- si la méthode définit un résultat de type `T`, alors on considère que l'appel de cette méthode produit une valeur de type `T`. Toute les restrictions habituelles pour les valeurs de type `T` s'appliquent. Considérons par exemple la méthode suivante :

  ```
  public static boolean a(int x) {
    corps de la méthode
  }
  ```

  Le résultat de la méthode est de type `boolean`. On pourra donc l'utiliser dans tout contexte valide pour un `boolean`. L'appel suivant est donc incorrect :

  ```
  int x=a(2);
  ```

  Il est en effet impossible de placer une valeur de type `boolean` dans une variable de type `int`.

Conséquence pour la programmation : l'instruction `return`

Quand on indique qu'une méthode définit un résultat, il faut **impérativement** utiliser dans la méthode une nouvelle instruction, le `return` (voir la section 6.4.5). Cette instruction a deux effets :

1. elle est suivie d'une expression : le résultat de la méthode est la valeur de cette expression (il est aussi possible d'utiliser l'instruction `return` seule dans une méthode sans résultat, comme l'explique la section 6.4.5) ;
2. elle provoque l'arrêt de la méthode et le retour dans la méthode appelée.

L'utilisation de `return` obéit à une contrainte importante : l'expression qui suit le `return` doit impérativement avoir pour type celui indiqué par la déclaration de la méthode.

Il est maintenant facile de comprendre l'exemple 6.13. L'instruction `return` de la ligne 3 a pour effet de définir le résultat de la méthode comme étant la valeur de l'expression `2*x+1`. Si on effectue l'appel `f(2)`, le paramètre formel `x` prend la valeur `2` et l'expression précédente vaut donc `5`. L'expression `f(2)` est donc remplacée par la valeur `5`.

Vocabulaire

Quand une méthode définit un résultat, on dit qu'elle **renvoie un résultat** et on désigne le résultat de la méthode par l'expression "**valeur renvoyée** par la méthode". Pour rester proche de l'anglais *return*, on dit parfois que la méthode **retourne un résultat** et on parle alors de "**valeur de retour** de la méthode".

6.4.4 Mécanisme du `return`

Étudions maintenant plus précisément le mécanisme de l'instruction `return`. Nous avons décrit à la section 6.3.3 l'exécution de l'appel d'une méthode en laissant en suspend le problème du résultat. Considérons donc un appel de méthode quelconque et son exécution par le processeur. Après les 3 premières étapes détaillées dans la section 6.3.3, le processeur commence à exécuter les instructions de la méthode :

- s'il ne rencontre jamais l'instruction `return`, c'est nécessairement que la méthode ne définit pas de résultat. Alors, quand l'exécution de toutes les instructions de la méthode est terminée, le processeur supprime toutes les variables de la méthode appelée et passe directement à l'instruction suivante de la méthode appelante (après avoir supprimé la barrière de séparation entre les zones mémoires) ;
- si la méthode définit une résultat, le processeur rencontre nécessairement une instruction `return`, qu'il exécute de la façon suivante :
  1. il commence par évaluer l'expression qui suit le mot clé `return` ;
  2. il conserve le résultat de cette évaluation dans une zone mémoire particulière ;
  3. il détruit toutes les variables de la méthode appelée ;
  4. il retourne dans la méthode appelante et supprime la barrière de séparation entre les zones mémoires ;
  5. il continue l'exécution de la méthode appelante.

  Si l'appel de la méthode est utilisé dans une expression, il est alors remplacé par le contenu de la zone mémoire particulière[10]. Cette zone mémoire réalise donc la transmission depuis la méthode appelée vers la méthode appelante.

Exemple 6.15 :

Analysons le comportement de l'exemple suivant :

Transmission

```
1  public class Transmission {
2    public static boolean estPair(int x) {
3      return x%2==0;
4    }
5    public static void main(String[] args) {
6      System.out.println(estPair(4));
7      boolean result=estPair(5)||estPair(6);
8      System.out.println(result);
9    }
10 }
```

Considérons l'exécution de la ligne 6 :

1. pour pouvoir exécuter l'appel de la méthode `println`, le processeur doit d'abord évaluer le paramètre de cette méthode, c'est-à-dire l'expression `estPair(4)`. Pour ce faire, il doit donc exécuter la méthode `estPair`. Il commence par évaluer le paramètre effectif de cette méthode, ce qui ne demande aucun calcul (c'est en fait la valeur littérale `4`) ;
2. le processeur exécute tout le mécanisme de passage de paramètre. On se retrouve avec une variable `x` dans la zone mémoire de la méthode `estPair`, initialisée à la valeur `4` (voir la figure 6.10) ;

[10] Pour simplifier, on représente en général cette zone mémoire comme une variable sans nom située dans la zone mémoire de la méthode appelante, voir par exemple la figure 6.12. Notons que notre représentation du passage de paramètre est tout aussi simplifiée.

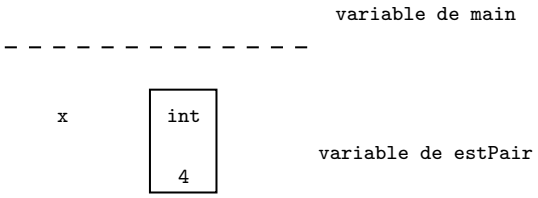

FIG. 6.10 – méthode `estPair`, juste avant la ligne 3

3. le processeur exécute la méthode `estPair`. Pour ce faire, il exécute la ligne 3 :
   (a) le processeur évalue l'expression qui suit le `return` et obtient ici `true` ;
   (b) il place cette valeur dans la zone mémoire réservée au résultat (voir la figure 6.11) ;

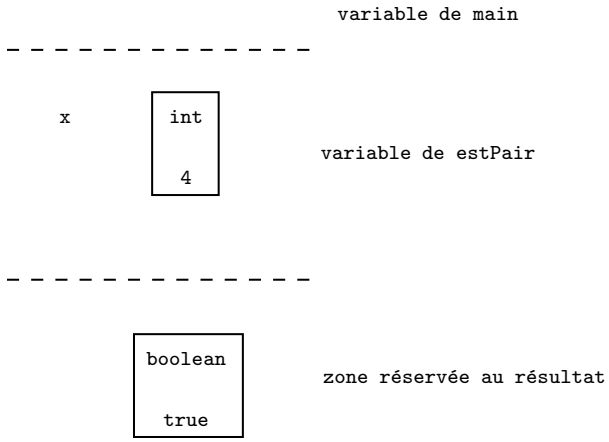

FIG. 6.11 – méthode `estPair`, pendant l'exécution de la ligne 3

   (c) il détruit l'unique variable de la méthode `estPair` (le paramètre formel `x`) ;
   (d) il retourne dans la méthode appelante ;
4. la valeur de l'expression `estPair(4)` est contenue dans la zone mémoire spéciale : c'est la valeur `true`. La figure 6.12 donne une représentation simplifiée de la situation dans la mémoire. En fait, la situation est légèrement plus complexe, mais l'approximation proposée convient parfaitement ;

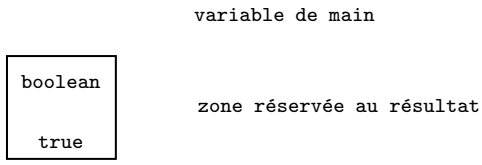

FIG. 6.12 – méthode `main`, juste après l'appel de `estPair(4)`

5. le processeur peut maintenant exécuter l'appel à la méthode `println`, ce qui a pour effet d'afficher la valeur `true`.

Le mécanisme est bien sûr le même pour les deux appels de la ligne 7. Finalement, le programme affiche :

AFFICHAGE

```
true
true
```

REMARQUE

La définition du résultat d'une méthode relève d'un mécanisme plus simple que le passage de paramètre. On peut résumer simplement l'action de `return` :

1. `return` *exp* ; définit le résultat de la méthode qui contient l'instruction comme étant la valeur de l'expression *exp*. On remplacera donc l'appel de la méthode par cette valeur ;
2. l'instruction `return` provoque l'arrêt *immédiat* de la méthode en cours d'exécution.

6.4.5 Le `return` est obligatoire

Principe de base

Il est très important de noter qu'une méthode qui définit un résultat (c'est-à-dire dont le type du résultat n'est pas `void`) doit nécessairement contenir **au moins un** `return`. Si on ne donne pas de `return`, le compilateur n'accepte pas la méthode, comme l'illustre l'exemple suivant :

Exemple 6.16 :

On considère la classe suivante :

PasDeReturn

```
1 public class PasDeReturn {
2   public static int f(int x) {
3     System.out.println(2*x+1);
4   }
5   public static void main(String[] args) {
6     int y=f(2);
7   }
8 }
```

Le compilateur refuse la méthode `f` et affiche le message d'erreur suivant :

ERREUR DE COMPILATION

```
PasDeReturn.java:2: missing return statement
  public static int f(int x) {
                             ^
1 error
```

Le compilateur indique donc très simplement qu'il manque une instruction `return`.

Il faut aussi que le type de l'expression qui suit le (ou les) `return`(s) d'une méthode soit compatible avec le type indiqué dans l'en-tête de la méthode. Comme le montre l'exemple suivant, le compilateur accepte qu'on utilise un type moins général que celui indiqué dans l'en-tête, mais il refuse bien entendu ce qu'il refuserait pour une affectation.

Exemple 6.17 :

On propose la classe suivante :

MauvaisType

```
1  public class MauvaisType {
2    public static double f(int x) {
3      return 2*x+1;
4    }
5    public static int g(int x) {
6      return x/2.0;
7    }
8    public static void main(String[] args) {
9      double y=f(2);
10     int z=g(3);
11   }
12 }
```

Le compilateur refuse le `return` de la ligne 6 et donne le message d'erreur suivant :

Erreur de compilation

```
MauvaisType.java:6: possible loss of precision
found   : double
required: int
    return x/2.0;
           ^
1 error
```

On voit donc que le compilateur accepte le `return` de la ligne 3, simplement car une valeur de type `int` (le type de l'expression qui suit le `return`) peut être placée dans une variable de type `double` (le type attendu pour le résultat de la méthode). Au contraire, la ligne 6 est refusée car une valeur de type `double` ne peut pas être placée dans une variable de type `int`.

Interaction avec les sélections et les boucles

Comme toujours en `Java`, le compilateur a une interprétation très restrictive des sélections et des boucles (voir à ce sujet les sections 4.2.3 et 5.5.2). Pour résumer les enseignements tirés des chapitres précédents, on peut dire que le compilateur ne tient pas compte du contenu des boucles `for` et `while` pour déterminer si une variable est initialisée avant son utilisation. Il ne tient pas non plus compte du contenu d'un `if` sans `else`. Pour les `if else`, il exige une initialisation par alternative.

Pour déterminer si une méthode contient bien un `return`, le compilateur va appliquer exactement les mêmes règles. Le but de ces règles n'est pas de s'assurer de la présence d'un `return` dans la méthode, mais plutôt d'être certain que quelles que soient les circonstances de l'exécution de la méthode (en particulier quelles que soient les valeurs des paramètres par exemple), le processeur va nécessairement rencontrer et donc exécuter une instruction `return`. Illustrons par un exemple les règles utilisées :

Exemple 6.18 :

On considère le programme suivant, dans lequel le programmeur tente de réaliser une méthode de calcul de la valeur absolue d'un entier :

MauvaisAbs

```
1  public class MauvaisAbs {
2    public static int abs(int x) {
3      if(x>=0) {
4        return x;
5      }
6      if(x<0) {
7        return -x;
8      }
9    }
10   public static void main(String[] args) {
11     System.out.println(abs(-2));
12   }
13 }
```

Le compilateur refuse la méthode et donne le message d'erreur suivant :

Erreur de compilation

```
MauvaisAbs.java:2: missing return statement
  public static int abs(int x) {
                               ^
1 error
```

On voit donc que le compilateur ne tient pas compte des `return`s des lignes 4 et 7. En effet, il ne peut pas savoir que les conditions `x>=0` et `x<0` couvrent tout les cas possibles et qu'on est donc certain qu'un `return` sera bien exécuté. Comme nous venons de l'écrire, un `return` dans un `if` seul n'est pas pris en compte dans la vérification qu'une méthode contient bien au moins un `return`.

REMARQUE

L'exemple précédent est volontairement mal programmé. Il est en effet clair qu'on doit simplement utiliser un `if else` à la place des deux `if`s. Le problème est alors réglé car chaque alternative du `if else` comprend bien un `return` (cf l'exemple 6.21). Le but de l'exemple précédent est d'illustrer simplement une erreur possible.

Nous verrons à la section 6.4.6 un exemple de problème potentiel quand on utilise une boucle dans une méthode (exemple 6.22).

Le cas des méthodes sans résultat

Dans une méthode qui ne définit pas de résultat (`void`), le `return` est *a priori* inutile. Si on tente de l'utiliser pour définir un résultat, le compilateur refuse le programme, comme le montre l'exemple suivant :

Exemple 6.19 :

On considère le programme suivant :

ReturnVoid

```
1  public class ReturnVoid {
2    public static void f(int x) {
3      return 2*x-1;
4    }
```

```
5   public static void main(String[] args) {
6     System.out.println(f(4));
7   }
8 }
```

Le compilateur refuse à la fois l'utilisation du `return` dans la méthode `f` et l'utilisation du "résultat" de cette méthode dans le `main` :

Erreur de compilation

```
ReturnVoid.java:3: cannot return a value from method whose result type is void
    return 2*x-1;
           ^
ReturnVoid.java:6: 'void' type not allowed here
    System.out.println(f(4));
                       ^
2 errors
```

Cependant, le `return` a deux effets. Si l'effet de définition du résultat n'a pas sens pour une méthode sans résultat, celui d'arrêt de l'exécution de la méthode peut être utile. On a donc le droit d'utiliser un `return` **sans expression** afin d'interrompre l'exécution d'une méthode. Voici un exemple d'utilisation :

Exemple 6.20 :

On considère le programme suivant :

ReturnStop

```
1  public class ReturnStop {
2    public static void f(int x,int y) {
3      for(int i=0;i<5;i++) {
4        if(2*x-i<y) {
5          return;
6        }
7        System.out.println(i);
8      }
9      System.out.println("Maximum");
10   }
11   public static void main(String[] args) {
12     f(4,5);
13     f(10,2);
14   }
15 }
```

Ce programme compile parfaitement et produit l'affichage suivant :

Affichage

```
0
1
2
3
0
1
```

```
2
3
4
Maximum
```

On constate que le `return` de la ligne 5 provoque l'arrêt de la méthode quand il est exécuté. C'est le cas dans l'appel `f(4,5)` car quand `i` prend la valeur 4, la condition `2*x-i<y` s'évalue en `true` et provoque donc l'exécution du contenu du `if`. Pour l'appel `f(10,2)`, la condition vaut toujours `false`. Le `return` n'est jamais exécuté et la méthode se termine donc après l'exécution de sa dernière instruction (ligne 9).

REMARQUE

Il faut se garder de confondre l'interruption d'une boucle par un `break` et l'interruption d'une méthode par un `return`. Dans le premier cas, seule la boucle qui contient le `break` est terminée. Dans le second au contraire, c'est toute la méthode qui se termine. Si on reprend l'exemple précédent et qu'on remplace le `return` par un `break`, on obtient le programme suivant :

BreakStop

```
1  public class BreakStop {
2    public static void f(int x,int y) {
3      for(int i=0;i<5;i++) {
4        if(2*x-i<y) {
5          break;
6        }
7        System.out.println(i);
8      }
9      System.out.println("Maximum");
10   }
11   public static void main(String[] args) {
12     f(4,5);
13     f(10,2);
14   }
15 }
```

L'affichage produit est différent :

AFFICHAGE

```
0
1
2
3
Maximum
0
1
2
3
4
Maximum
```

Dans cette nouvelle version, le `break` provoque l'arrêt de la boucle `for`. De ce fait, la ligne 9 est toujours exécutée, même si le `break` l'est. Tout appel à la méthode `f` provoque au minimum l'affichage de `Maximum`. Ce n'est bien sûr pas le cas dans la version qui utilise un `return`.

6.4.6 Plusieurs `returns`

Une méthode qui définit un résultat doit comporter au moins un `return`. Rien n'empêche d'utiliser plusieurs `return` si cela permet de rendre le programme plus lisible et/ou plus efficace. Reprenons l'exemple 6.18 qui tente de calculer la valeur absolue d'un entier. On peut proposer une solution correcte utilisant un seul `return` ou une solution plus claire utilisant deux `returns` :

Exemple 6.21 :

Commençons par la solution avec un seul `return`. Nous nous contentons ici de donner la méthode :

AbsUnReturn

```
1  public static int abs(int x) {
2    int r;
3    if(x>=0) {
4      r=x;
5    } else {
6      r=-x;
7    }
8    return r;
9  }
```

Le principe est simple, on commence par calculer le résultat qu'on place dans la variable `r`. Ensuite, on renvoie le contenu de cette variable. C'est souvent la meilleure solution, en particulier quand le calcul du résultat est complexe.

Dans le cas du calcul de la valeur absolue, les opérations sont très simples. On peut donc utiliser deux `returns`, ce qui rend le programme plus simple à lire :

AbsDeuxReturn

```
1  public static int abs(int x) {
2    if(x>=0) {
3      return x;
4    } else {
5      return -x;
6    }
7  }
```

Le programme est plus simple à lire car le résultat des différentes alternatives apparaît directement dans le traitement de chacune d'elle. On voit que le résultat est `x` lui-même quand cette valeur est positive et `-x` dans le cas contraire. Pour arriver à la même conclusion dans la version avec un `return`, il faut être attentif aux évolutions de la variable `r`. Il est donc parfois plus simple d'utiliser plusieurs `returns`.

Nous avons vu à la section 5.6 qu'il parfois très utile d'interrompre une boucle au milieu de son exécution. On peut ainsi obtenir des programmes plus clairs. Comme le `return` interrompt la méthode qui le contient, on peut l'utiliser pour terminer une boucle. Revenons par exemple sur le problème de la recherche de la première apparition d'un chiffre dans un nombre (présenté à la section 5.6.1).

Exemple 6.22 :

On souhaite écrire une méthode qui à deux entiers `n` et `c` associe la position de la première occurrence de `c` dans `n` (en numérotant les chiffres de droite à gauche et en démarrant à 0). Si `c` n'apparaît pas dans `n`, la méthode devra renvoyer l'entier `-1`. On peut proposer une solution simple en s'inspirant de la solution de l'exemple 5.19, ce qui donne la solution suivante :

positionChiffre

```
1  public static int position(int n,int c) {
2    int pos=0;
3    while(n>0) {
4      if(n%10==c) {
5        return pos;
6      } else {
7        pos++;
8        n/=10;
9      }
10   }
11   return -1;
12 }
```

Le `return` de la ligne 5 correspond à la découverte d'une occurrence de `c` dans `n`. Dès que cette occurrence est découverte, on peut arrêter l'exécution de la méthode et renvoyer la valeur obtenue.

Le deuxième `return` (ligne 11) est tout aussi important puisqu'il traite le cas où aucune occurrence de `c` n'est découverte dans `n`. Dans ce cas, comme convenu, la méthode renvoie la valeur `-1`. Remarquons que sans ce `return`, la méthode n'est pas acceptée par le compilateur. En effet, le `return` à l'intérieur de la boucle `while` n'est pas pris en compte par le compilateur : la boucle peut éventuellement ne pas s'exécuter.

On pourrait bien sûr calquer plus fidèlement sur la solution de l'exemple 5.19, c'est-à-dire calculer la position puis renvoyer le résultat, en utilisant un seul `return`. La solution qui vient d'être proposée est plus claire et plus directe.

Il est très important de comprendre qu'**une méthode s'arrête dès que le processeur exécute un `return`**. Quand une méthode possède plusieurs `return`s, c'est bien entendu le premier rencontré qui est exécuté, ce qui provoque l'arrêt de la méthode. Pour bien fixer les idées, considérons un exemple :

Exemple 6.23 :

On souhaite écrire une méthode qui calcule le `k`-ième chiffre d'un entier strictement positif (numérotation de droite à gauche, en partant de 0). Si un tel chiffre n'existe pas, la méthode doit renvoyer `-1`. Un programmeur propose la solution **fausse** suivante :

ChiffreFaux

```
1  /* Ce programme ne fait pas ce qu'il est censé faire... */
2  public class ChiffreFaux {
3    public static int chiffre(int n,int k) {
4      for(int i=0;i<=k;i++) {
5        if(n>0) {
6          return n%10;
7        }
8        n/=10;
9      }
10     return -1;
```

```
11    }
12    public static void main(String[] args) {
13      System.out.println(chiffre(23245,1));
14      System.out.println(chiffre(34248,0));
15      System.out.println(chiffre(33241,5));
16    }
17  }
```

L'affichage produit par le programme est le suivant :

AFFICHAGE

```
5
8
1
```

On constate que le résultat n'est absolument pas celui souhaité. Le problème principal provient du `return` dans la boucle (ligne 6). Le programmeur peut croire que le dernier `return` exécuté sera le bon, c'est-à-dire que le `return` n'arrête pas l'exécution de la méthode. Si c'était le cas, la solution serait en effet correcte : le dernier `return` correspondrait effectivement au chiffre recherché.

Cependant, le `return` arrête l'exécution de la boucle. De ce fait, la méthode proposée renvoie toujours le chiffre de position zéro (le plus à droite) de `n`. Voici maintenant une solution **correcte** :

ChiffreOk

```
1   public class ChiffreOk {
2     public static int chiffre(int n,int k) {
3       for(int i=0;i<k;i++) {
4         n/=10;
5       }
6       if(n>0) {
7         return n%10;
8       } else {
9         return -1;
10      }
11    }
12    public static void main(String[] args) {
13      System.out.println(chiffre(23245,1));
14      System.out.println(chiffre(34248,0));
15      System.out.println(chiffre(33241,5));
16    }
17  }
```

L'affichage produit est celui souhaité, à savoir :

AFFICHAGE

```
4
8
-1
```

REMARQUE

Il existe une variante de l'erreur étudiée dans l'exemple précédent : on pourrait croire que le `return` ne termine pas la méthode et accumule les résultats. Après une boucle qui exécuterait plusieurs `return`s (insistons sur le fait que c'est **strictement impossible**), on obtiendrait comme résultat la "liste" des valeurs renvoyées par les `return`s. **Cette vision naïve est complètement erronée.**

Il existe effectivement un moyen de faire renvoyer une liste de valeurs par une méthode, et plus généralement de manipuler des listes de valeurs, mais cela passe par la notion de tableau, qui n'a aucun rapport avec d'hypothétiques `return`s multiples. Nous étudierons les tableaux au chapitre 8.

6.5 Méthodologie

6.5.1 Introduction

Les méthodes posent un problème nouveau pour l'apprenti programmeur. Les méthodes introduisent de nouveaux détails techniques (passage de paramètres, résultat, etc.), mais il ne faut pas laisser cet aspect masquer l'essentiel : les méthodes sont avant tout un moyen d'organiser et de simplifier un programme. Il est parfaitement possible en théorie de programmer sans utiliser de méthodes. Pourtant, cela ne viendrait à l'idée d'aucun programmeur expérimenté : les méthodes sont indispensables si on souhaite écrire rapidement des programmes faciles à maintenir.

La difficulté principale est qu'il est pratiquement impossible d'expliquer pourquoi utiliser une méthode serait une bonne idée dans un programme donné. Seule l'expérience permet de choisir judicieusement les méthodes à programmer pour réaliser une application de qualité. Le but de cette section est de proposer quelques conseils permettant une utilisation correcte des méthodes, sous forme de règles informelles. Ces règles sont issues d'une assez longue expérience pratique et sont acceptées de façon relativement universelle. Ceci étant, elles restent imprécises, floues et très difficiles à justifier. Elles constituent une aide pour le programmeur débutant mais ne remplacent pas l'expérience...

La section 6.6 sera consacrée à un élément méthodologique important : l'utilisation de plusieurs classes. Dans la présente section, nous nous contenterons donc d'étudier le cas d'une ou plusieurs méthodes dans une seule classe.

6.5.2 Algorithmes et méthodes

Les méthodes et les algorithmes sont très fortement liés. En fait, on peut dire qu'une méthode est en général la réalisation en `Java` d'un algorithme. Un algorithme est constitué de deux parties : la description (les données et les résultats) et les étapes. On retrouve exactement la même structure dans une méthode : l'en-tête décrit les paramètres de la méthode ainsi que son résultat, alors que le contenu de la méthode programme les étapes du traitement réalisé par celle-ci.

De plus, nous avons vu dans la présentation des algorithmes du chapitre 4 qu'un algorithme ne s'occupe pas de saisie et d'affichage, ce qui doit être en général le cas des méthodes, sauf bien entendu la méthode `main` qui se charge en général de l'interaction avec l'utilisateur du programme (cf les sections 6.5.3 et 6.5.4 pour des détails). C'est en fait cette similitude qui est la plus importante. Un programme considéré comme un tout est en général beaucoup plus complexe qu'un algorithme, car il doit effectuer des saisies, contrôler qu'elles sont valides, enchaîner plusieurs traitements et présenter le résultat final. Au contraire, une méthode considérée isolément se charge de résoudre un petit problème, à partir de données supposées correctes. Elle produit un résultat sous forme d'une valeur, dont elle ne réalise pas l'affichage.

Il existe cependant une différence fondamentale entre algorithme et méthode : dans un algorithme, on peut proposer plusieurs résultats, de natures différentes. Comme nous l'avons vu au chapitre 4 (plus précisément à la section 4.3.4), il possible d'écrire par exemple un algorithme de résolution d'une équation du second degré. Le résultat de l'algorithme est alors soit un réel, soit deux réels, soit pour finir deux complexes. Or, il est difficile d'écrire une méthode qui renvoie comme résultat parfois un réel, parfois deux réels et parfois deux complexes. C'est même impossible en utilisant les outils étudiés jusqu'à présent. De façon générale, un algorithme ne comporte pas d'élément technique et est en fait une version abstraite d'une méthode. Celle-ci devra parfois être beaucoup plus complexe que l'algorithme. Dans certains cas, il sera opportun de programmer un algorithme sous forme de plusieurs méthodes.

6.5.3 Affichage et/ou résultat ?

L'affichage joue un rôle assez particulier dans les "vrais" programmes. Dans l'écrasante majorité des programmes proposés dans le présent ouvrage, nous avons utilisé des affichages (grâce à la méthode `println`), afin d'illustrer le comportement des constructions étudiées. Dans les programmes professionnels, l'interaction avec l'utilisateur passe très rarement par ce genre d'affichage : elle s'effectue par l'intermédiaire d'une interface graphique, c'est-à-dire de fenêtres, boutons, menus, boites de dialogue, etc. Ces dispositifs sont assez complexes à utiliser et ne fonctionnent en général pas avec la méthode `println` (nous les étudierons de façon très succincte au chapitre 10).

On peut donc se demander s'il est légitime de réaliser un affichage (par une technique ou une autre) dans une méthode. En effet, si une méthode produit un résultat, il n'est pas judicieux d'afficher ce résultat *au sein de la méthode elle-même*. Si la méthode affiche son résultat, on doit choisir quand on la programme une technique d'affichage donné et on ne peut plus en changer sans modifier la méthode. Au contraire, si la méthode ne fait aucun affichage, on peut toujours utiliser son résultat pour l'afficher en utilisant une technique quelconque. De ce fait, la solution sans affichage est plus souple et offre plus de possibilités que la solution avec affichage.

On peut donc énoncer la règle suivante : **en général, une méthode qui produit un résultat ne doit pas contenir d'affichage**. De façon plus restrictive, **seules les méthodes qui ont pour unique but de réaliser un affichage doivent contenir des instructions d'affichage**.

Pour bien comprendre les différences entre l'affichage dans et en dehors d'une méthode, étudions un exemple simple :

Exemple 6.24 :

On considère une méthode de calcul de $n!$:

Factorielle

```
1  public class Factorielle {
2    public static long fact(int n) {
3      long r=1;
4      for(int i=2;i<=n;i++) {
5        r*=i;
6      }
7      return r;
8    }
9    public static long factAff(int n) {
10     long r=1;
11     for(int i=2;i<=n;i++) {
12       r*=i;
13     }
14     System.out.println(r);
```

```
15      return r;
16    }
17    public static void main(String[] args) {
18      System.out.println(fact(10));
19      System.out.println(factAff(8));
20      fact(7);
21      factAff(9);
22      System.out.println(2*fact(6));
23      System.out.println(2*factAff(6));
24    }
25  }
```

Les quatre premières lignes de la méthode `main` (lignes 18 à 21) peuvent laisser croire que la méthode `factAff`, qui calcule $n!$ et affiche le résultat avant de le renvoyer, est plus facile d'utilisation que `fact` qui se contente de calculer le résultat.

Les deux dernières lignes montrent que malgré les apparences, la méthode `fact` est la plus utile : si on souhaite tout simplement afficher $2(5!)$, on ne peut pas vraiment utiliser `factAff` qui va d'abord afficher $5!$ avant de nous laisser utiliser le résultat pour calculer $2(5!)$ et afficher la valeur obtenue.

L'affichage produit par le programme est le suivant :

AFFICHAGE

```
3628800
40320
40320
362880
1440
720
1440
```

REMARQUE

Il faut se garder de comprendre ces règles de façon inversée. On pourrait croire en effet qu'elles incitent à écrire des méthodes sans résultat, et à se contenter d'afficher les éléments produits par la méthode directement dans celle-ci. Il faut comprendre **exactement le contraire**. Il faut on contraire s'efforcer d'écrire des méthodes qui produisent des résultats. On peut ainsi réutiliser le travail produit par une méthode afin d'obtenir des résultats plus complexes. Les règles proposées indiquent simplement qu'il ne faut pas que des méthodes qui produisent un résultat affichent ce résultat.

6.5.4 Saisie dans les méthodes

La situation de la saisie est très proche de celle de l'affichage. Dans une application professionnelle, les saisies s'effectuent par l'intermédiaire d'une interface graphique. Pour simplifier l'apprentissage du langage `Java`, nous avons développé un ensemble de méthodes simples d'utilisation qui permettent d'interagir aisément avec l'utilisateur (voir la section 3.5.4). Comme pour l'affichage, les saisies proposées par les méthodes `readXxxx` de la classe `Console` sont assez éloignées des solutions basées sur une interface graphique.

On peut donc énoncer une règle assez similaire à celles proposées pour les affichages : **seules les méthodes qui ont pour unique but de réaliser une saisie doivent contenir des instructions de saisie.**

Commençons par un exemple qui ne respecte pas cette règle :

Exemple 6.25 :

Reprenons le calcul de la factorielle :

FactorielleSaisie

```
1  import dauphine.util.*;
2  public class FactorielleSaisie {
3    public static long fact(int n) {
4      long r=1;
5      for(int i=2;i<=n;i++) {
6        r*=i;
7      }
8      return r;
9    }
10   public static long factSaisie() {
11     int n=Console.readInt();
12     long r=1;
13     for(int i=2;i<=n;i++) {
14       r*=i;
15     }
16     return r;
17   }
18   public static void main(String[] args) {
19     Console.start();
20     System.out.println(factSaisie());
21     System.out.println(fact(Console.readInt()));
22     System.out.println(fact(2*Console.readInt()));
23   }
24 }
```

La méthode `factSaisie` effectue une saisie afin d'obtenir l'entier dont elle va calculer la factorielle. De prime abord, la méthode semble plus simple à utiliser que `fact` qui demande une saisie pour chaque appel. Cependant, la ligne 22 montre que `factSaisie` est très limitée : il est impossible par exemple calculer la factorielle du double d'un entier saisi.

Il reste parfaitement légitime d'écrire des méthodes qui contiennent des saisies, à condition que le seul but de ces méthodes soit justement de réaliser une saisie, par exemple en tenant compte de certaines contraintes :

Exemple 6.26 :

La méthode `readInt`, de la classe `Console`, permet de saisir un entier quelconque. On peut proposer la méthode suivante qui effectue la saisie d'un nombre entier strictement positif :

readPositiveInt

```
1  public static int readPositiveInt() {
2    int result;
3    do {
4      result=Console.readInt();
5      if(result<=0) {
```

```
6       System.out.println("La valeur saisie doit être strictement positive");
7     }
8   } while(result<=0);
9   return result;
10 }
```

6.5.5 Modifications des paramètres d'une méthode

Nous avons vu que la déclaration des paramètres d'une méthode déclare en fait des variables propres à cette méthode dont les valeurs initiales sont celles des paramètres effectifs lors de l'appel de la méthode. Comme les paramètres formels sont des variables, on peut donc les modifier avant de les utiliser. Dans la mesure du possible, on s'interdira ce genre de modifications :

Exemple 6.27 :

Considérons la méthode suivante :

modif1

```
1 public static double f(double x) {
2   x=x+1;
3   return 2*Math.sin(x);
4 }
```

L'incrémentation de `x` est ici parfaitement inutile car on peut écrire :

```
return 2*Math.sin(x+1);
```

Par contre, si on doit calculer $(x+1)\sin(x+1)$, il est parfaitement justifié d'incrémenter d'abord `x` puis de faire `return x*Math.sin(x) ;`. En effet, dans ce dernier cas, on ne calcule *qu'une fois* `x+1` alors que l'écriture `return (x+1)*Math.sin(x+1) ;` provoque *deux fois* le calcul de cette valeur.

De façon générale, la modification des paramètres doit être justifiée par des considérations pratiques ou techniques. Une erreur courante quand on n'a pas compris que le paramètre n'est pas une variable classique mais qu'il sert à communiquer avec les autres méthodes du programme, est de saisir une valeur pour le paramètre dans la méthode, comme par exemple :

modif2

```
1 public static double f(double x) {
2   x=Console.readDouble();
3   return 2*Math.sin(x);
4 }
```

C'est parfaitement **inutile** (et même **stupide**). En effet, la valeur initiale de `x` est ici perdue et donc la méthode n'a en fait reçu *aucune* information provenant de la méthode qui l'a appelée. Donc la déclaration de paramètre était inutile et `x` aurait dû être déclarée comme une variable de la méthode. Par exemple :

modif3

```
1 public static double f() {
2   double x;
3   x=Console.readDouble();
4   return 2*Math.sin(x);
5 }
```

REMARQUE

Il y a une différence fondamentale entre la *modification* d'un paramètre qui est parfois justifiée et l'*écrasement* d'un paramètre qui traduit une **incompréhension profonde de la notion de paramètre**.

Notons de plus que la solution proposée (qui consiste à considérer `x` comme variable de la méthode plutôt que comme paramètre) n'est pas la bonne : il faudrait bien sûr supprimer la saisie.

Il est très important de bien comprendre la notion de paramètre : les paramètres servent à transmettre des informations depuis la méthode appelante vers la méthode appelée. Voici un exemple de mauvaise utilisation des paramètres :

Exemple 6.28 :

On considère le programme suivant :

Somme

```
1  public class Somme {
2    public static int sigma(int n,int s) {
3      s=0;
4      for(int i=1;i<=n;i++) {
5        s+=i;
6      }
7      return s;
8    }
9    public static void main(String[] args) {
10     System.out.println(sigma(5,0));
11     System.out.println(sigma(10,12));
12   }
13 }
```

La méthode `sigma` calcule la somme des entiers de 1 à `n` (son premier paramètre) et renvoie cette valeur. Pourquoi le programmeur a-t-il ajouté à la méthode `sigma` le paramètre `s` ? Simplement parce qu'il a confondu la notion de paramètre avec celle de variable. La méthode `sigma` a bien entendu besoin d'une variable afin de réaliser son travail. Mais elle n'a aucunement besoin d'un second paramètre. De ce fait, la bonne version de la méthode est donnée dans le programme suivant :

SommeOk

```
1  public class SommeOk {
2    public static int sigma(int n) {
3      int s=0;
4      for(int i=1;i<=n;i++) {
5        s+=i;
6      }
7      return s;
8    }
9    public static void main(String[] args) {
10     System.out.println(sigma(5));
11     System.out.println(sigma(10));
12   }
13 }
```

On remarque que les appels de la méthode `sigma` (lignes 10 et 11) sont maintenant plus naturels : ils demandent un seul paramètre.

6.6 Plusieurs classes

6.6.1 Motivations

Nous avons vu à la section 6.1.3 qu'une application `Java` est constituée de *plusieurs* classes. Pour l'instant, nous sommes contenté d'exemples utilisant une seule classe. Dans la pratique, tous les programmes utiles sont constitués de nombreuses classes. Diverses raisons justifient l'utilisation de plusieurs classes :

- la compilation d'un programme `Java` prend du temps. Or, quand on modifie une classe, on doit seulement recompiler le texte de cette classe, même si elle s'insère dans une application constituée de nombreuses classes. En découpant correctement l'application en plusieurs classes, on peut réduire très simplement le temps nécessaire à la compilation et donc à la programmation ;
- le découpage en plusieurs classes permet d'organiser l'application en plusieurs éléments relativement indépendants. C'est le meilleur moyen de travailler en groupe, chaque programmeur étant responsable de plusieurs classes, les autres programmeurs se contentant d'utiliser les classes en question, sans pouvoir les modifier.

Il existe d'autres raisons à l'utilisation de nombreuses classes. Nous étudierons par exemple au chapitre 9 une justification très importante. Remarquons au passage que nous évoquons ici à la fois une motivation très pratique (temps de compilation) qui apparaît de façon évidente dès qu'on dépasse le stade des programmes d'une dizaine de lignes, et une motivation d'ordre méthodologique (division du travail) qui ne peut apparaître que dans le cadre d'un travail plus professionnel.

6.6.2 Nom complet et nom simple

Il y a essentiellement deux "difficultés" liées à l'écriture d'applications basées sur plusieurs classes :

1. il faut impérativement utiliser les noms complets des méthodes (on se retrouve donc dans la situation du chapitre 3 où on utilisait des méthodes déjà définies) ;
2. toutes les classes qui constituent une application doivent être stockées dans le même dossier sur l'ordinateur utilisé[11].

Voici un exemple un peu pathologique, mais qui illustre bien la différence entre un nom complet et un nom simple :

Exemple 6.29 :

On considère tout d'abord la classe suivante :

First

```
1  public class First {
2    public static int f(int x) {
3      return 2*x-1;
4    }
5    public static int g(int a,int b) {
6      return f(a+b)*f(a-b);
7    }
8  }
```

[11] Nous simplifions ici volontairement la situation. Les classes situées dans un même dossier font partie du même paquet (cf 3.7) et peuvent être utilisées sans importation. L'interaction entre la programmation "abstraite" en `Java` et l'aspect pratique est parfois délicate. Le lecteur intéressé par les détails techniques pourra se référer à [7] ou encore à [6], en particulier la section 7 et plus spécifiquement le point 7.2.

Comme elle ne comporte pas de `main`, ce n'est pas la classe principale de l'application. On remarque que la méthode `g` est définie à partir de la méthode `f`, en utilisant un nom simple, car les deux méthodes appartiennent à la même classe. Considérons maintenant la classe principale :

Second

```
1  public class Second {
2    public static int f(int x) {
3      return x/2;
4    }
5    public static void main(String[] args) {
6      System.out.println(f(5));
7      System.out.println(First.f(5));
8      System.out.println(First.g(2,1));
9    }
10 }
```

On remarque que cette classe comporte un appel avec un nom simple (ligne 6) et deux appels avec des noms complets. Ils sont relativement simples à interpréter :

- l'appel `f(5)` utilise un nom simple : il fait donc référence à la méthode `f` définie par la classe `Second` ;
- l'appel `First.f(5))` utilise un nom complet : il fait donc référence à la méthode `f` définie par la classe `First` ;
- l'appel `First.g(5))` utilise un nom complet : il fait donc référence à la méthode `g` définie par la classe `First`. Quand cette méthode est exécutée, le processeur considère qu'il est "dans" la classe de définition de la méthode (ici la classe `First`). De ce fait, les appels simples de `f` de la ligne 6 de la méthode `g` ne posent aucun problème : c'est bien la méthode `f` de la classe `First` qui sera appelée.

L'affichage produit par le programme est le suivant :

Affichage

```
2
9
5
```

6.6.3 Une application réaliste : la dichotomie

Présentation du problème

Le problème à résoudre est le suivant : étant donnés une fonction continue f de $\mathbb{R}$ dans $\mathbb{R}$ et deux réels $a < b$ tels que $f(a)f(b) < 0$, nous cherchons un $x_0 \in]a, b[$ tel que $f(x_0) = 0$. Il s'agit donc de trouver une solution de l'équation $f(x) = 0$.

Commençons par quelques remarques mathématiques :

- la condition $f(a)f(b) < 0$ signifie que $f(a)$ et $f(b)$ sont de signes opposés et tous deux non nuls ;
- le théorème des valeurs intermédiaires (applicable car f est continue) et la condition précédente permettent d'affirmer qu'il existe au moins un $u \in]a, b[$ tel que $f(u) = 0$, et donc que le problème possède au moins une solution ;
- il n'y a aucune raison pour que f possède une seule racine dans l'intervalle $]a, b[$, donc nous ne trouverons qu'une partie de la solution.

Algorithme d'une méthode de résolution

Pour trouver une racine de l'équation $f(x) = 0$, nous allons utiliser une technique qui consiste à réduire peu à peu l'intervalle dans lequel on est sûr qu'une solution existe. Quand l'intervalle sera suffisamment petit, le milieu de celui-ci sera une bonne approximation de la solution. Plus précisément, si on obtient un intervalle $[u, v]$ tel que $f(u)f(v) < 0$, on sait que $f(x) = 0$ possède une solution dans $]u, v[$. Si $v - u < \epsilon$, on en déduit que u ou v sont des approximations de la solution à ϵ près[12], ce qui est suffisant en général avec par exemple $\epsilon = 10^{-10}$.

Comment peut-on réduire l'intervalle dans lequel on est sûr qu'il existe une racine ? En fait c'est relativement simple. Supposons par exemple qu'on parte d'un intervalle $]a_0, b_0[$ pour lequel $f(a_0)f(b_0) < 0$. Étudions alors la valeur de f au point $c = \frac{a_0+b_0}{2}$. Trois cas sont possibles :

1. $f(c) = 0$:
   Dans ce cas, on a découvert la solution, et on a donc gagné.
2. $f(c)$ est du même signe que $f(a_0)$:
   Dans ce cas, $f(c)f(b_0) < 0$ (car $f(b_0)$ et $f(a_0)$ sont de signes opposés), ce qui signifie (grâce au théorème des valeurs intermédiaires) que $f(x) = 0$ possède une solution dans l'intervalle $]c, b_0[$.
3. $f(c)$ est du même signe que $f(b_0)$:
   Dans ce cas, $f(c)f(a_0) < 0$ (car $f(b_0)$ et $f(a_0)$ sont de signes opposés), ce qui signifie (grâce au théorème des valeurs intermédiaires) que $f(x) = 0$ possède une solution dans l'intervalle $]a_0, c[$.

Donc, sauf dans le cas très particulier où $f(c) = 0$, on passe d'un intervalle $]a_0, b_0[$ à un intervalle deux fois plus petit, $]a_1, b_1[$, qui est constitué de la moitié gauche ou de la moitié droite de l'intervalle de départ. Ce procédé est illustré par la figure 6.13. L'algorithme de résolution, appelée résolution par dichotomie, est alors le suivant :

Données :
- f la fonction dont on doit trouver un zéro ;
- a et b les bornes de l'intervalle de départ ;
- ϵ, la précision avec laquelle on veut connaître la racine.

Résultat : un réel x tel que $f(x) \simeq 0$ (plus précisément, un réel x tel qu'il existe un réel $x_0 \in]x - \epsilon, x + \epsilon[$ tel que $f(x_0) = 0$).

1. placer dans `u` et `v` respectivement a et b.
2. répéter les instructions suivantes :
   (a) placer dans `m` le milieu de [`u`, `v`]
   (b) si f(`m`) est du même signe que f(`u`), placer `m` dans `u`
   (c) si f(`m`) est du même signe que f(`v`), placer `m` dans `v`
   (d) continuer la boucle si f(`m`) $\neq 0$ **et** si `v-u` est supérieur à ϵ.
3. Résultat : `m`.

Le principe de l'algorithme est simple : il consiste à utiliser la technique de réduction de l'intervalle de recherche de la racine jusqu'à ce que la racine soit découverte ou que l'intervalle soit de longueur très petite. On est sûr d'obtenir un intervalle de longueur inférieur à $\epsilon > 0$ un réel quelconque car on divise la longueur par deux à chaque étape, ce qui implique que cette longueur tend vers zéro. La figure 6.14 illustre par un organigramme l'algorithme obtenu.

[12] Il s'agit ici d'une précision absolue, ce qui est notoirement moins significatif qu'une précision relative de la forme $\frac{v-u}{v+u}$. Nous utilisons la précision absolue pour simplifier le programme.

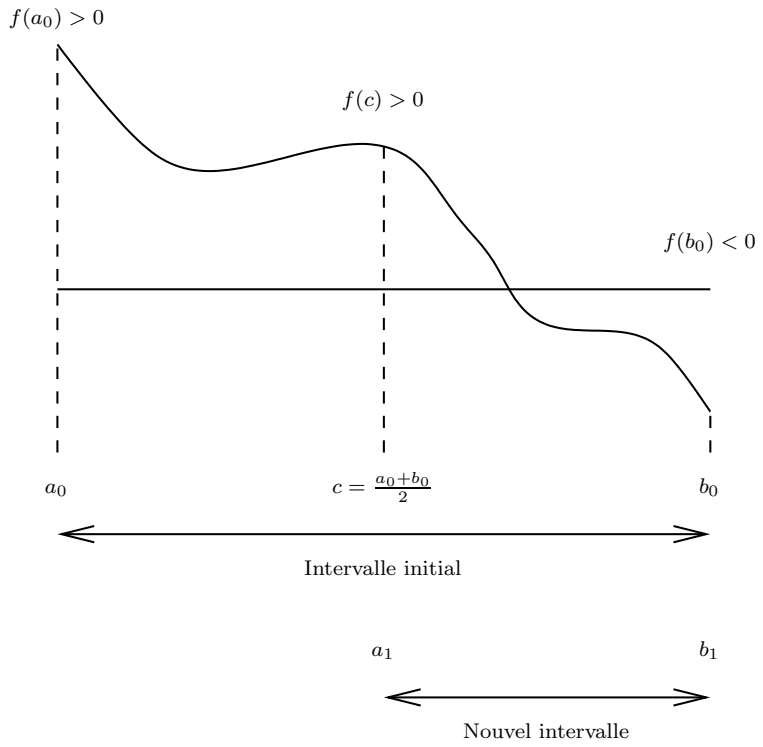

FIG. 6.13 – Réduction de l'intervalle de recherche en dichotomie

Programme de la solution

Il n'existe pas de méthode simple permettant de stocker une fonction dans une variable[13]. De ce fait, on ne peut pas traduire tel que l'algorithme proposé. Le moyen le plus simple de programmer la résolution dichotomique est de représenter la fonction étudiée par une méthode. Pour pouvoir changer rapidement et efficacement de fonction, nous plaçons la méthode dans une classe à part :

Fonction

```
1 public class Fonction {
2   public static double f(double x) {
3     /* un exemple de fonction */
4     return Math.sin(2*x);
5   }
6 }
```

L'algorithme de dichotomie est alors programmé dans une autre classe :

Dichotomie

```
1 public class Dichotomie {
2   public static double root(double u,double v,double precision) {
3     // la méthode de recherche
4     double milieu,fMilieu;
5     double fU=Fonction.f(u),fV=Fonction.f(v);
6     do {
7       milieu=(u+v)/2.0;
```

[13] C'est bien sûr possible, mais cela demande la mise en œuvre de mécanismes complexes, en particulier le sous-typage (cf [10]).

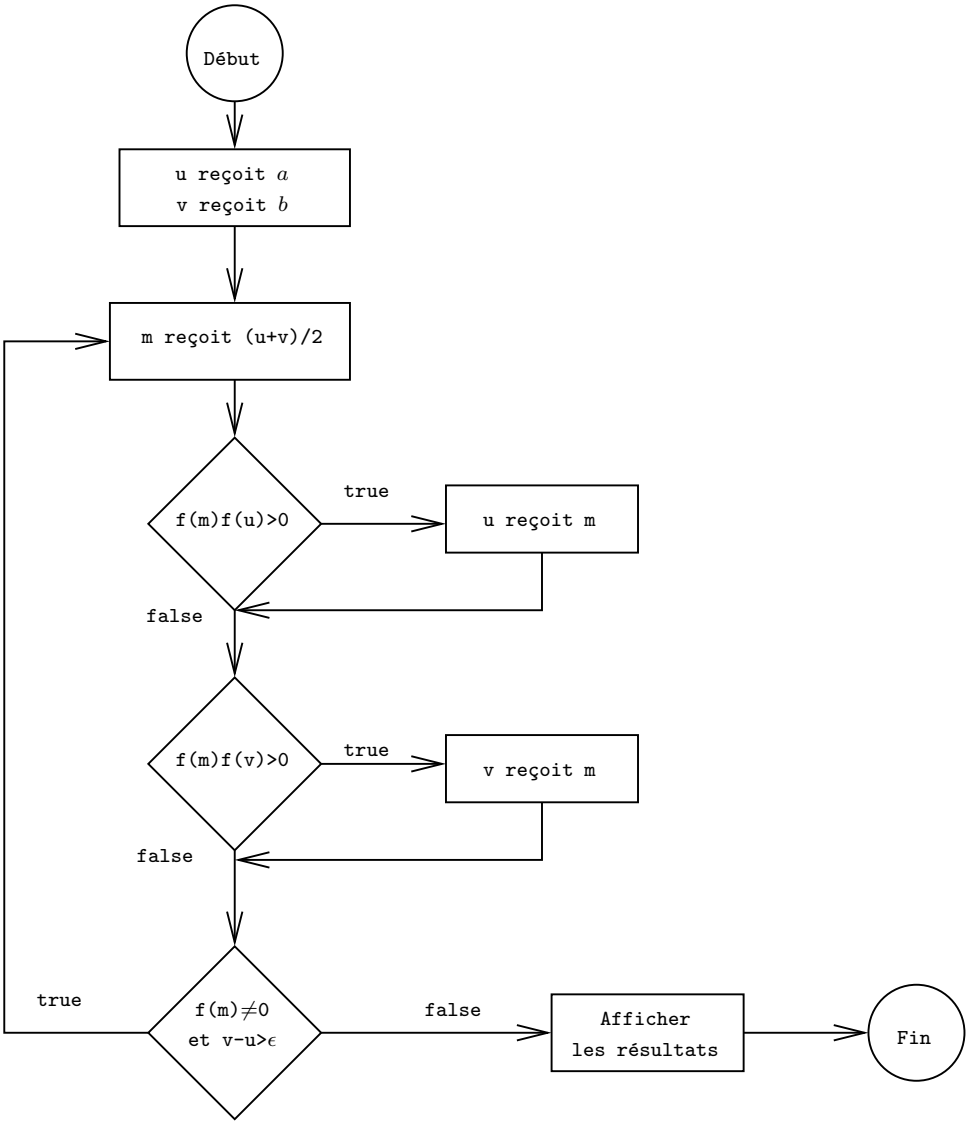

FIG. 6.14 – Organigramme de la dichotomie

```
8       fMilieu=Fonction.f(milieu);
9       if (fMilieu*fU>0) {
10        // f(milieu) du même signe que f(u)
11        u=milieu;
12        fU=fMilieu;
13      } else if (fMilieu*fV>0) {
14        // f(milieu) du même signe que f(v)
15        v=milieu;
16        fV=fMilieu;
17      }
18    } while ((fMilieu != 0.0) && ((v-u)>precision));
19    return milieu;
20  }
21 }
```

Enfin, on utilise une classe principale qui se contente d'effectuer les saisies et d'afficher les résultats :

Resolution

```
1  import dauphine.util.*;
2  public class Resolution {
3    public static void main(String[] args) {
4      Console.start();
5      // lecture des paramètres essentiels
6      System.out.print("Borne inférieure : ");
7      double u=Console.readDouble();
8      System.out.print("Borne supérieure : ");
9      double v=Console.readDouble();
10     System.out.print("Précision : ");
11     double epsilon=Console.readDouble();
12     double resultat;
13     resultat=Dichotomie.root(u,v,epsilon);
14     System.out.println("Racine :"+resultat);
15     System.out.println("Valeur de la fonction :"+Fonction.f(resultat));
16   }
17 }
```

Il est bien entendu possible de regrouper les trois méthodes dans une seule classe, mais cela n'a aucun intérêt, alors que la programmation de la fonction à étudier sous forme d'une méthode d'une classe complètement indépendante des autres permet de ne recompiler que la classe en question quand on souhaite changer de fonction.

6.6.4 La catégorie `private`

Principe

Nous avons vu à la section 6.1.2 la forme générale d'une déclaration de méthode. Comme pour tout élément d'une classe, chaque méthode possède une catégorie. Nous nous sommes pour l'instant limité à la catégorie `public`. Dès qu'on réalise de gros programmes, on est amené à utiliser la catégorie `private`. Quand une méthode est déclarée `private`, **elle n'est utilisable que dans les méthodes de la même classe**. Considérons un exemple simple :

Exemple 6.30 :

On considère la classe suivante :

AvecPrivate

```
1 public class AvecPrivate {
2   private static int f(int a,int b) {
3     return 2*a+b;
4   }
5   public static int g(int a,int b) {
6     return (f(a,b)+f(b,a))/2;
7   }
8 }
```

La méthode `g` est `public`. Elle pourra donc être utilisée par n'importe quelle autre méthode. Par contre, la méthode `f` est `private`. Elle ne pourra donc être utilisée que dans la classe `AvecPrivate`, comme le fait par exemple la méthode `g`. Si on tente d'utiliser les méthodes dans une autre classe, le compilateur détecte toute tentative d'accès interdit. Considérons par exemple la classe suivante :

UsePrivate

```
1 public class UsePrivate {
2   public static void main(String[] args) {
3     System.out.println(AvecPrivate.g(2,3));
4     System.out.println(AvecPrivate.f(1,2));
5   }
6 }
```

L'appel de la ligne 3 ne pose aucun problème. Par contre, le compilateur refuse l'appel de la ligne 4 car la méthode `f` est `private`. Il donne le message d'erreur suivant :

Erreur de compilation

```
UsePrivate.java:4: f(int,int) has private access in AvecPrivate
    System.out.println(AvecPrivate.f(1,2));
                                   ^
1 error
```

Intérêt

Il est assez difficile de justifier l'intérêt de la catégorie `private`. De prime abord, cette catégorie a pour seul effet de limiter l'utilisation d'une méthode, sans rien apporter en plus de la catégorie `public`. Nous rencontrons ici un problème crucial de la programmation : l'expérience prouve qu'il est parfois très utile de limiter les possibilités des programmeurs. Cela rend les programmes plus simples à interpréter et donc plus simples à maintenir. Quand on travaille à plusieurs sur un gros programme, un programmeur peut avoir intérêt à "cacher" certaines parties de son travail (certaines méthodes). Le programmeur est en effet confronter à deux contraintes contradictoires :

1. en général, on a intérêt à multiplier les méthodes. En exagérant un peu, on peut dire que la longueur maximale d'une méthode doit être qu'une vingtaine de lignes. Dès qu'une méthode devient trop longue, on a en général intérêt à la casser en plusieurs méthodes ;
2. en général, chaque partie d'un programme doit avoir le moins de lien possible avec les autres parties (ce qui permet aux programmeurs de travailler indépendamment les uns des autres). Or, chaque méthode est un lien potentiel entre deux parties d'un programme.

La seule solution permettant de concilier les deux contraintes est d'utiliser la catégorie `private`. Le programmeur peut multiplier les méthodes sans craintes : seules les méthodes `public` seront utilisables par les autres programmeurs. Le programmeur d'une partie peut donc choisir exactement les liens possibles entre les autres parties du programme et son propre travail.

6.7 Conseils d'apprentissage

Ce chapitre contient deux difficultés. Du point de vue technique, l'écriture de méthodes demande la compréhension de deux mécanismes complexes : celui du passage de paramètre et celui de la définition du résultat. Du point de vue pratique, l'écriture de méthodes demande une certaine expérience : il n'est pas rare de commencer par écrire des méthodes inutiles ou mal conçues. Pour un meilleur apprentissage, nous suggérons de procéder comme suit :

- Il est impossible d'apprendre à écrire des méthodes si on ne maîtrise pas parfaitement l'utilisation de méthodes déjà définies. Il est donc intéressant de relire le chapitre 3 avant tout nouvel apprentissage.
- Il est indispensable de comprendre et de maîtriser **les mécanismes de l'appel de méthode** :
  1. le point le plus délicat concerne le **passage de paramètre**. Le mécanisme est assez complexe et le **passage par valeur** a des conséquences qui peuvent paraître déroutantes ;
  2. la **définition du résultat** d'une méthode pose moins de problèmes, mais il faut bien retenir que le `return` **termine la méthode**.
- Quand les éléments techniques sont connus et maîtrisés, on peut s'intéresser à la partie méthodologique : comment écrire des méthodes utiles. Il est tout à fait contre productif de travailler la méthodologie tant qu'on ne maîtrise pas l'aspect technique. De plus, il faut noter que seule l'expérience permet de bien utiliser les méthodes. Le seul moyen d'acquérir cette expérience est de programmer... Pour faciliter les premiers pas, on peut cependant s'appuyer sur les recommandations proposées dans le chapitre :
  – les affichages et les saisies doivent être bannis des méthodes, sauf dans le cas où l'unique but de la méthode est de réaliser un affichage ou une saisie ;
  – les paramètres servent uniquement à la communication entre la méthode appelante et la méthode appelée : en général, il n'y a pas lieu de modifier leur contenu ;
  – l'utilisation de plusieurs classes permet de travailler plus efficacement, surtout quand on programme à plusieurs.

CHAPITRE 7

Première approche des objets

Sommaire

Introduction

Dans les chapitres précédents, nous avons appris à utiliser les types fondamentaux en tant qu'outils de représentation des informations à faire manipuler par un programme. Nous connaissons les outils algorithmiques nécessaires à l'écriture de programmes évolués (les sélections et les boucles), et nous savons grâce aux méthodes et aux classes comment organiser un programme complexe. Il nous reste maintenant à améliorer la représentation de l'information dans un programme pour dépasser les applications simplistes qui manipulent quelques nombres entiers ou réels.

Comment faire par exemple pour manipuler un texte ? Comment stocker dans la mémoire de l'ordinateur les notes d'un élève (par exemple) si le nombre de notes peut être saisi par l'utilisateur *au moment de l'exécution du programme* ? De façon plus générale, comment obtenir une représentation simple des informations à faire traiter par un programme ?

Le but de ce chapitre est d'introduire la notion **d'objet** qui généralise la notion de valeur et qui permet la représentation simple et efficace d'informations complexes. Nous commencerons par étudier le **type objet** `String` qui permet la manipulation de chaînes de caractères, c'est-à-dire de suite quelconque de caractères. Cela nous permettra d'avoir un premier aperçu de la notion d'objet.

Nous introduirons ensuite les **méthodes d'instance** qui permettent de manipuler les objets et jouent en fait le rôle joué par les opérateurs pour les types fondamentaux. Nous étudierons ensuite les problèmes que posent la gestion mémoire des objets en parlant de la **manipulation par référence** et de ces conséquences complexes.

Motivés par un souci d'efficacité, nous présenterons un exemple d'objets modifiables, les `StringBuffer`s qui proposent une autre représentation des chaînes de caractères. Nous utiliserons cet exemple important pour illustrer les conséquences les plus subtiles de la manipulation par référence à savoir les **effets de bord**.

7.1 Les chaînes de caractères : le type `String`

7.1.1 Le type `String` et les valeurs littérales associées

Principe de base

Commençons par une définition :

Définition 7.1 *Une **chaîne de caractères** est une suite finie de caractères.*

Nous avons vu qu'il existe un type fondamental `char` qui permet de manipuler **un** caractère. Le problème est donc de manipuler **plusieurs** caractères en même temps.

Un des moyens pour manipuler une chaîne de caractères en `Java` est de passer par le type `String` (nous verrons qu'on peut aussi utiliser le type `StringBuffer`, cf la section 7.4). Ce nouveau type peut s'utiliser exactement comme n'importe quel autre type (pour l'instant, nous ne connaissons que les types fondamentaux, présentés à la section 2.1.2) :

- on peut déclarer des variables de type `String` ;
- une méthode peut demander un ou plusieurs paramètres de type `String` (voir l'exemple 7.5) ;
- une méthode peut déclarer `String` comme type pour son résultat (voir les exemples 7.20 et 7.25).

Valeur littérale

Nous avons déjà rencontré des *valeurs littérales* de type `String`. En effet, quand on veut afficher un texte à l'écran, on écrit `System.out.println("Bonjour")`, ce qui provoque l'affichage du texte compris entre les guillemets (ici, `Bonjour`). Ce texte est une chaîne de caractères et le compilateur `Java` lui associe le type `String`. Les valeurs littérales de type `String` sont donc les suites de caractères quelconques, comprises entre des guillemets.

Exemple 7.1 :

Chaque ligne suivante comporte une valeur littérale de type `String` :

```
"Bonjour"
"x+2"
"static void main("
"Voilà un texte..."
""
```

Notons que la dernière valeur littérale désigne la chaîne de caractères vide, c'est-à-dire ne contenant aucun caractère. Cette construction n'est pas possible avec le type `char` pour lequel les valeurs littérales doivent toujours désigner exactement un caractère. La construction ■ est de ce fait refusée par le compilateur.

Les caractères spéciaux

On peut éventuellement vouloir insérer un guillemet dans une chaîne de caractères. Si on écrit `"exemple " "`, on obtient pas une chaîne de caractères. Pour pouvoir mettre un guillemet dans un chaîne de caractères, il faut le faire précéder d'un *antislash*[1], c'est à dire l'écrire `\"`. Notre exemple précédent s'écrit donc `"exemple \" "`. Il faut être attentif à bien terminer les chaînes de caractères et à ne pas mettre de guillemet directement dans une chaîne car le compilateur a des difficultés à comprendre un programme qui comporte des erreurs liées aux chaînes, comme l'illustre l'exemple suivant :

[1] un *backslash* en anglais.

Exemple 7.2 :

On considère le programme suivant :

MauvaiseValeur

```
1 public class MauvaiseValeur {
2   public static void main(String[] args) {
3     String s="ABCD " EFGH ";
4   }
5 }
```

Le compilateur refuse le programme et donne des messages d'erreur qui montrent que la présence du guillemet sans antislash perturbe son analyse du programme :

Erreur de compilation

```
MauvaiseValeur.java:3: ';' expected
    String s="ABCD " EFGH ";
                     ^
MauvaiseValeur.java:3: unclosed string literal
    String s="ABCD " EFGH ";
                          ^
2 errors
```

Notons d'autre part que certains autres "caractères" ne peuvent pas être insérés directement dans une valeur littérale de type `String`. On ne peut pas par exemple passer à la ligne à l'intérieur d'une chaîne, comme l'indique l'exemple suivant :

Exemple 7.3 :

On considère le programme suivant :

PasDeSautDeLigne

```
1 public class PasDeSautDeLigne {
2   public static void main(String[] args) {
3     String tentative="un passage à
4 la ligne";
5     System.out.println(tentative);
6   }
7 }
```

Le programme est refusé par le compilateur qui donne le message d'erreur suivant :

Erreur de compilation

```
PasDeSautDeLigne.java:3: unclosed string literal
    String tentative="un passage à
                     ^
PasDeSautDeLigne.java:4: unclosed string literal
la ligne";
        ^
2 errors
```

La tabulation (qui permet d'aligner proprement des textes) n'est pas non plus directement utilisable en `Java`. Pour obtenir ces "caractères" spéciaux, on utilise de nouveau le symbole *antislash*, selon la correspondance dans le tableau qui suit.

Séquence	“caractère” correspondant
\n	passage à la ligne
\t	tabulation
\\	le caractère \ lui-même
\’	le caractère ’ (type `char`)

Voici un exemple simple d’application :

Exemple 7.4 :

Nous allons écrire un programme qui affiche le tableau proposé, en le plaçant d’abord dans une `String`. Pour que le programme soit lisible, on utilise par anticipation l’opérateur `+` (cf la section 7.1.2), qui colle bout à bout deux `String`s :

Echappement

```
1  public class Echappement {
2    public static void main(String[] args) {
3      String tableau="Séquence\t‘‘caractère’’ correspondant\n";
4      tableau+="\\n\t\tpassage à la ligne\n";
5      tableau+="\\t\t\ttabulation\n";
6      tableau+="\\\\\t\tle caractère \\ lui-même\n";
7      tableau+="\\’\t\tle caractère ’ (type char)";
8      System.out.println(tableau);
9      char c='\’';
10     System.out.println(c);
11   }
12 }
```

Les deux dernières lignes du programme réalisent une démonstration de la séquence `\’`, inutile pour les `String`s, mais indispensable pour le type fondamental `char`. L’affichage produit par le programme est le suivant :

Affichage

```
Séquence        ‘‘caractère’’ correspondant
\n              passage à la ligne
\t              tabulation
\\              le caractère \ lui-même
\’              le caractère ’ (type char)
```

L’utilisation de la tabulation permet un alignement correct des colonnes. Chaque colonne peut contenir au plus huit caractères, d’où l’utilisation de deux tabulations dans l’exemple proposé.

Une application

Voici un exemple simple d’application des `String`s :

Exemple 7.5 :

Ce programme calcule la moyenne de deux notes :

Moyenne

```
1  import dauphine.util.*;
2  public class Moyenne {
3    public static int lireNote(String message) {
4      int result;
```

```
5       System.out.print(message);
6       result=Console.readInt();
7       while (result<0||result>20) {
8         System.out.println("La valeur doit être comprise entre 0 et 20");
9         System.out.print(message);
10        result=Console.readInt();
11      }
12      return result;
13    }
14    public static void main(String[] args) {
15      Console.start();
16      int noteDeMath,noteDEco;
17      noteDeMath=lireNote("Note de math=");
18      noteDEco=lireNote("Note d'économie=");
19      System.out.println("Moyenne= "+((double)noteDeMath+noteDEco)/2);
20    }
21  }
```

On remarque que la méthode `lireNote` prend pour paramètre une chaîne de caractères, qui est ensuite transmise à `System.out.print` pour être affichée. Ceci permet d'utiliser la méthode `lireNote` pour lire diverses notes en indiquant comme paramètre effectif le message à afficher pour demander la note. Une utilisation de ce programme provoque typiquement ce genre d'affichage :

AFFICHAGE

```
Note de math=34
La valeur doit être comprise entre 0 et 20
Note de math=15
Note d'économie=-5
La valeur doit être comprise entre 0 et 20
Note d'économie=13
Moyenne= 14.0
```

L'utilisation d'un paramètre de type `String` permet donc de proposer une méthode de lecture plus pratique à utiliser.

7.1.2 Concaténation

Dans le chapitre 2, nous avons appris à utiliser les différents types fondamentaux en étudiant les opérateurs applicables aux valeurs de ces différents types. Le seul opérateur utilisable avec les `String`s est le symbole de l'addition, qui réalise une opération de concaténation. On peut écrire par exemple `String s="Bon"+"jour";`. La variable `s` contiendra alors la chaîne de caractères `"Bonjour"`. Dans cette concaténation, on peut bien sûr faire apparaître des variables de type `String`. On peut de plus ajouter plusieurs chaînes les unes aux autres dans une même expression, comme par exemple `String s="Bonjour"+" à "+"tous";`.

L'intérêt principal de la concaténation est de simplifier l'utilisation des méthodes d'affichage, comme nous l'avons déjà vu à la section 3.5.3. En fait, nous manipulons depuis ce chapitre des `String`s, mais sans le savoir !

De plus, comme nous l'avons justement vu lors de notre apprentissage de l'affichage, il est en fait possible de concaténer une valeur d'un type fondamental quelconque à une `String`. Le processeur

va d'abord convertir la valeur en une `String` (cf la section 7.1.4) puis concaténer les deux chaînes. La chaîne résultat de `12+"ABC"` est donc `"12ABC"`.

La concaténation nous permet d'améliorer le programme de l'exemple 7.5 :

Exemple 7.6 :

On propose le programme suivant :

MoyenneConcat

```
1  import dauphine.util.*;
2  public class MoyenneConcat {
3    public static int lireValeur(String message,int min,int max) {
4      int result;
5      System.out.print(message);
6      result=Console.readInt();
7      while (result<min||result>max) {
8        System.out.println("La valeur doit être comprise entre "+min
9                           +" et "+max);
10       System.out.print(message);
11       result=Console.readInt();
12     }
13     return result;
14   }
15   public static void main(String[] args) {
16     Console.start();
17     int noteDeMath,noteDEco;
18     noteDeMath=lireValeur("Note de math=",0,20);
19     noteDEco=lireValeur("Note d'économie=",0,20);
20     System.out.println("Moyenne= "+((double)noteDeMath+noteDEco)/2);
21   }
22 }
```

Dans cette nouvelle version, la méthode `lireValeur` est très générale : elle correspond à une saisie dans laquelle on impose un intervalle pour la valeur entière proposée par l'utilisateur.

REMARQUE

L'opération de concaténation permet de construire progressivement, par ajout successif, un résultat complexe, en particulier au moyen d'une boucle. Nous illustrerons ce type d'utilisation dans l'exemple 7.20.

7.1.3 Saisie

On peut demander à l'utilisateur de saisir une chaîne de caractères sous la forme d'une `String`, en utilisant la méthode `readString` de la classe `Console`. Voici un exemple très élémentaire d'utilisation de cette méthode :

Exemple 7.7 :

Un programme qui s'essaye à la politesse :

Bonjour

```
1  import dauphine.util.*;
2  public class Bonjour {
3    public static void main(String[] args) {
```

```
4     Console.start();
5     System.out.print("Donnez votre nom : ");
6     String nom=Console.readString();
7     System.out.println("Bonjour, "+nom);
8   }
9 }
```

Ce programme donnera par exemple l'affichage suivant :

```
Donnez votre nom : Toto
Bonjour, Toto
```

7.1.4 Conversion

D'un type quelconque vers une `String`

On a vu dans les exemples précédents, et, plus généralement, depuis le début du présent ouvrage, que l'ordinateur est capable de transformer automatiquement une valeur numérique en une chaîne de caractères lors d'une concaténation. De façon générale, on peut se demander comment transformer une valeur d'un type fondamental en une `String` de façon directe, sans passer par une concaténation.

On remarque tout d'abord qu'une approche naïve n'est pas valide, comme le montre l'exemple suivant :

Exemple 7.8 :

Considérons le programme suivant :

MauvaisType

```
1 public class MauvaisType {
2   public static void main(String[] args) {
3     double x=2;
4     String s=x;
5     String t=2;
6   }
7 }
```

Le compilateur refuse le programme et affiche le messages d'erreur suivants :

Erreur de compilation

```
MauvaisType.java:4: incompatible types
found   : double
required: java.lang.String
    String s=x;
             ^
MauvaisType.java:5: incompatible types
found   : int
required: java.lang.String
    String t=2;
             ^
2 errors
```

On voit donc que le compilateur refuse de convertir un entier et un réel en une `String`.

REMARQUE

Le type `String` se comporte donc comme les autres types, c'est-à-dire que seules certaines affectations sont considérées comme valides, celles qui respectent les types. Dans le cas de `String`, on peut seulement placer dans une variable de type `String` des éléments du même type.

Comme la solution naïve n'est pas utilisable, on utilise des méthodes de conversion. Il faut savoir que le type `String` est associé à une classe du même nom. Cette classe propose des méthodes de classe `valueOf` qui transforment leur paramètre en une chaîne de caractères (de type `String`).

Exemple 7.9 :

Voici comment le programme de l'exemple 7.8 peut être modifié pour devenir correct :

TypeCorrect

```
1  public class TypeCorrect {
2    public static void main(String[] args) {
3      double x=2;
4      String s=String.valueOf(x);
5      String t=String.valueOf(2);
6    }
7  }
```

Comme par exemple pour certaines méthodes de la classe `Math` (voir la section 3.3.2), il existe en fait une version de `valueOf` pour chaque type fondamental, ce qui permet de faire toutes les conversions souhaitées.

Contrairement à ce qu'on pourrait croire, le compilateur fait une différence entre un caractère seul et une chaîne de caractères (représentée par une `String`). Il est obligatoire de passer par la méthode `valueOf` adaptée pour pouvoir transformer un `char` en une `String`, comme l'illustre l'exemple suivant :

Exemple 7.10 :

On considère le programme suivant :

CharEtStringFaux

```
1  public class CharEtStringFaux {
2    public static void main(String[] args) {
3      char c='a';
4      String s="b";
5      s=c;
6      c=s;
7      s='u';
8      c="v";
9    }
10 }
```

Le compilateur refuse le programme et donne le message d'erreur suivant :

Erreur de compilation

```
CharEtStringFaux.java:5: incompatible types
found   : char
required: java.lang.String
    s=c;
      ^
CharEtStringFaux.java:6: incompatible types
```

```
found   : java.lang.String
required: char
    c=s;
      ^
CharEtStringFaux.java:7: incompatible types
found   : char
required: java.lang.String
    s='u';
      ^
CharEtStringFaux.java:8: incompatible types
found   : java.lang.String
required: char
    c="v";
      ^
4 errors
```

Les lignes 3 et 4 sont naturellement acceptées. Ce n'est pas le cas pour les lignes suivantes, car on tente soit de placer une `String` dans une variable de type `char` (lignes 6 et 8), soit le contraire (lignes 5 et 7). Grâce à la méthode `valueOf` de la classe `String`, il est possible de corriger les lignes 5 et 7, qu'on peut remplacer par :

```
s=String.valueOf(c);
s=String.valueOf('u');
```

Pour les lignes 6 et 8, il faut passer par une *méthode d'instance* qu'on étudiera à la section 7.2.3.

REMARQUE

Il est impossible de réaliser une conversion par une "mise entre guillemets" aux effets magiques. Supposons qu'on souhaite par exemple convertir en `String` le contenu de la variable `x` (de type `double` par exemple). Certains programmeurs débutants pensent qu'il suffit d'écrire `"x"` et que, *de façon exceptionnelle*, le compilateur comprendra qu'il faut en fait remplacer cette valeur littérale de type `String` par `String.valueOf(x)`. Cette supposition est incorrecte : **le compilateur n'interprète jamais le contenu d'une valeur littérale de type `String`**. La chaîne `"x"` reste immuablement la chaîne `"x"`, même s'il existe une variable appelée `x`.

D'une `String` vers un type fondamental

Que penser de la chaîne de caractères `"1234"` ? On peut se demander si le processeur est capable de comprendre que cette chaîne correspond à un entier. La réponse est à la fois négative et positive. Du côté négatif, comme nous venons de le rappeler dans la remarque précédente, le compilateur n'interprète jamais le contenu d'une chaîne de caractères, ce qui est illustré par l'exemple suivant :

Exemple 7.11 :

Le programme suivant tente naïvement de placer une chaîne de caractères contenant l'écriture décimale d'un entier dans une variable de type `int` :

StringNoInt

```
1 public class StringNoInt {
2   public static void main(String[] args) {
3     String s="1234";
```

```
4       int x=s;
5     }
6   }
```

Le compilateur refuse ce programme et donne le message d'erreur suivant :

Erreur de compilation

```
StringNoInt.java:4: incompatible types
found   : java.lang.String
required: int
    int x=s;
          ^
1 error
```

Par contre, il existe des méthodes de conversion. En d'autres termes, on peut dans un programme demander au processeur d'analyser le contenu d'une `String` afin d'en tirer éventuellement une valeur numérique.

Chaque type fondamental est associé à une classe portant presque le même. Nous avons d'ailleurs étudié certains éléments de ces classes au chapitre 3 (par exemple aux sections 3.4.3 et 3.6.3). Chacune de ces classes définie une méthode `parseXxx` dont le principe est de transformer la `String` paramètre en une valeur du type associé à la classe, quand cela est possible. En cas de problème, la méthode provoque l'arrêt du programme. Voici la liste des méthodes utilisables :

- la classe `Byte` définit la méthode :]

  `byte parseByte(String s)`

  Renvoie le contenu de la chaîne `s` sous forme d'un `byte`, si cela est possible. Produit une erreur sinon.

- la classe `Double` définit la méthode :

  `double parseDouble(String s)`

  Renvoie le contenu de la chaîne `s` sous forme d'un `double`, si cela est possible. Produit une erreur sinon.

- la classe `Float` définit la méthode :

  `float parseFloat(String s)`

  Renvoie le contenu de la chaîne `s` sous forme d'un `float`, si cela est possible. Produit une erreur sinon.

- la classe `Integer` définit la méthode :

  `int parseInt(String s)`

  Renvoie le contenu de la chaîne `s` sous forme d'un `int`, si cela est possible. Produit une erreur sinon.

- la classe `Long` définit la méthode :

  `long parseLong(String s)`

  Renvoie le contenu de la chaîne `s` sous forme d'un `long`, si cela est possible. Produit une erreur sinon.

- la classe `Short` définit la méthode :

  `short parseShort(String s)`

  Renvoie le contenu de la chaîne `s` sous forme d'un `short`, si cela est possible. Produit une erreur sinon.

Voici un exemple d'utilisation de certaines des méthodes :

Exemple 7.12 :

DemoParse

```
1  public class DemoParse {
2    public static void main(String[] args) {
3      String s="1234";
4      int x=Integer.parseInt(s);
5      System.out.println(x);
6      System.out.println(2*x);
7      s=String.valueOf(x/3.0);
8      double y=Double.parseDouble(s);
9      float z=Float.parseFloat(s);
10     System.out.println(y);
11     System.out.println(z);
12     System.out.println(s);
13   }
14 }
```

Ce programme produit l'affichage suivant :

AFFICHAGE

```
1234
2468
411.3333333333333
411.33334
411.3333333333333
```

Voici maintenant un exemple d'utilisation problématique :

Exemple 7.13 :

On considère le programme suivant :

DemoParseErreur

```
1  import dauphine.util.*;
2  public class DemoParseErreur {
3    public static void main(String[] args) {
4      Console.start();
5      String s=Console.readString();
6      int x=Integer.parseInt(s);
7      System.out.println(2*x);
8    }
9  }
```

Si l'utilisateur propose valeur entière, le programme fonctionne parfaitement et donne par exemple l'affichage suivant :

AFFICHAGE

```
56765
113530
```

Par contre, si l'utilisateur ne donne pas un entier, le programme plante et donne le message suivant :

Erreur d'exécution

```
2.5
Exception in thread "main" java.lang.NumberFormatException: 2.5
        at java.lang.Integer.parseInt(Integer.java:414)
        at java.lang.Integer.parseInt(Integer.java:454)
        at DemoParseErreur.main(DemoParseErreur.java:6)
```

Il est donc souhaitable d'utiliser les méthodes `readXxxx` adaptées plutôt que de tenter une conversion[2].

7.1.5 La notion d'objet

Quand on manipule les types fondamentaux, on travaille avec des valeurs. Tous les autres types sont des **types objet**. On ne parle plus alors de valeur du type considéré, mais d'**objet**. Par exemple, la chaîne de caractères `"exemple"` est un **objet** de type `String`.

Les objets sont une généralisation de la notion de valeur :

- chaque objet possède un type qui est nécessairement associé à une classe (voir la section 7.2.4), comme par exemple le type `String` qui correspond à une classe de même nom. On dit qu'**un objet est instance de la classe** correspondant à son type. On parlera donc d'objet instance de `String`. On dit aussi qu'un tel objet est de classe `String` ;
- pour chaque type objet, il existe un moyen de "fabriquer" un objet qui correspond aux valeurs littérales pour les types fondamentaux. Pour les `String`s, on dispose de valeurs littérales, mais c'est un cas exceptionnel. Pour les autres types objet, on utilisera un **constructeur** (cf la section 7.4.3) ;
- pour chaque type objet, il existe l'équivalent des opérateurs numériques et logiques : ce sont les **méthodes d'instance** (cf la section 7.2). Ces méthodes permettent de réaliser des opérations sur les objets, d'une façon analogue aux combinaisons de valeurs réalisées par les opérateurs pour les types fondamentaux. La concaténation des `String`s est un cas exceptionnel, les autres types objets n'utilisant pas en général d'opérateurs ;
- les objets sont gérés en mémoire d'une façon très différente de celle utilisée pour les types fondamentaux, ce qui permet un comportement beaucoup plus riche, mais souvent plus délicat à analyser (cf la section 7.3).

Dans la suite de ce chapitre, nous allons étudier les objets et les mécanismes spécifiques qui permettent leur manipulation, en nous basant au départ sur l'exemple des `String`s.

7.2 Les méthodes d'instance

7.2.1 Introduction

Les objets sont utilisables à travers de *méthodes* qui permettent des manipulations évoluées. De la même façon que les types fondamentaux peuvent être utilisés avec des opérations qui leur sont propres, les **méthodes d'instance** permettent de définir des opérations propres à un type objet donné. Les méthodes d'instance (qu'on peut aussi appeler **méthodes d'objet**) ne s'utilisent pas de la même façon que les méthodes de classe que nous avons étudiées aux chapitres 3 et 6. Comme les

[2] Les méthodes de la classe `Console` sont basées sur les méthodes que nous venons de présenter. La principale différence est que les méthodes `readXxxx` gèrent les erreurs.

méthodes de classe, les méthodes d'instance sont associées à une classe, celle du type objet auquel elles sont associées.

Dans cette section, nous allons étudier comment manipuler plus complètement les `String`s et apprendre par ce biais à utiliser des méthodes d'instance.

7.2.2 Principe

Un exemple

Avant d'étudier précisément les méthodes d'instance, commençons par observer un exemple simple :

Exemple 7.14 :

On considère le programme suivant :

DemoLength

```
1 public class DemoLength {
2   public static void main(String[] args) {
3     String s="ABCD";
4     System.out.println(s.length());
5     System.out.println("UVW".length());
6   }
7 }
```

Ce programme affiche :

AFFICHAGE

```
4
3
```

On constate donc que `s.length()` a pour valeur le nombre de caractères de la chaîne représentée par `s`. `length` est une méthode d'instance de la classe `String` et s'applique donc aux objets de type `String`.

Une méthode d'instance

La définition d'une **méthode d'instance** est strictement identique à celle d'une méthode de classe (cf la section 3.1.1) : c'est une suite d'instructions, désignée par un identificateur et élément d'une classe. Elle s'appelle avec des paramètres et produit éventuellement un résultat.

La seule différence entre les deux catégories de méthodes est qu'**une méthode d'instance possède obligatoirement un paramètre**. Ce paramètre est **l'objet appelant** et est obligatoirement du type[3] de la classe qui définit la méthode.

De plus, l'objet appelant n'est pas utilisé comme un paramètre classique : un appel de méthode d'instance se réalise en donnant un objet, suivi d'un point, suivi de l'identificateur de la méthode appelée (avec éventuellement des paramètres). Si on reprend l'exemple, l'appel `s.length()` obéit à ces nouvelles règles : `s` est une variable de type `String` et contient[4] donc un objet de type `String`. Cet objet est suivi d'un point et de l'identificateur de la méthode appelée.

[3] En fait, son type doit être un sous-type de celui de la classe qui définit la méthode (cf [10]), mais ce problème dépasse le cadre de cet ouvrage.

[4] Nous employons le mot contient pour simplifier la présentation. La section 7.3 montre que la situation est plus complexe.

REMARQUE

Nous avons déjà vu des exemples de méthodes d'instance : les méthodes `print` et `println` (cf la section 3.5.2). En effet, `System.out` est une constante de la classe `System`. Cette constante "contient" un objet de type `PrintStream` qui définit des méthodes d'instance `print` et `println`. L'écriture `System.out.println` est donc tout simplement un appel de méthode d'instance.

Compilation et exécution de l'appel

L'appel d'une méthode d'instance est traité exactement comme celui d'une méthode de classe. La seule nouveauté est l'objet appelant. Comme nous l'avons dit dans les paragraphes précédents, cet objet doit être du type de la classe qui définit la méthode d'instance. Cette règle est vérifiée à la compilation, comme le montre l'exemple suivant :

Exemple 7.15 :

Dans le programme suivant, le programmeur tente d'appeler la méthode `length` de différentes façons erronées :

BadLength

```
1 public class BadLength {
2   public static void main(String[] args) {
3     char c='A';
4     System.out.println(c.length());
5     System.out.println(String.length("ABCD"));
6     System.out.println(String.length());
7     System.out.println(12.length());
8   }
9 }
```

Le compilateur refuse toutes les utilisations envisagées et donne les messages d'erreur suivants :

Erreur de compilation

```
BadLength.java:7: ')' expected
    System.out.println(12.length());
                           ^
BadLength.java:4: char cannot be dereferenced
    System.out.println(c.length());
                       ^
BadLength.java:5: length() in java.lang.String cannot be applied to
(java.lang.String)
    System.out.println(String.length("ABCD"));
                              ^
BadLength.java:6: non-static method length() cannot be referenced from a
static context
    System.out.println(String.length());
                              ^
4 errors
```

On remarque que la ligne 7 pose un gros problème au compilateur qui ne comprend vraiment pas ce que le programmeur tente de faire (appliquer une méthode `length` à un entier). Les autres messages d'erreur sont assez délicats à interpréter :

- le message "*char cannot be dereferenced*" est l'expression qu'emploie le compilateur pour indiquer que `char` n'est pas un type objet (c'est un type fondamental) ;
- le message correspondant à la ligne 5 indique qu'il n'existe pas de méthode[5] appelée `length` et prenant comme paramètre un objet de type `String` ;
- enfin, le message correspondant à la ligne 6 indique que la méthode `length` est une méthode d'instance (c'est la signification de "*non-static method length()*") et qu'elle ne peut donc pas être appelée comme une méthode de classe (c'est la signification de "*referenced from a static context*").

Les règles concernant les signatures des méthodes (cf la section 3.3) s'appliquent pleinement. La méthode d'instance `length` (de la classe `String`) a comme signature `length()` : elle doit donc être appelée sans paramètre (excepté bien sûr l'objet appelant). L'exemple suivant montre qu'il n'est pas possible de faire autrement :

Exemple 7.16 :

On considère le programme :

BadLength2

```
1 public class BadLength2 {
2   public static void main(String[] args) {
3     System.out.println("ABCD".length(2));
4     System.out.println("ABCD".length("UV"));
5   }
6 }
```

Le compilateur refuse le programme et donne les messages d'erreur suivant :

Erreur de compilation

```
BadLength2.java:3: length() in java.lang.String cannot be applied to (int)
    System.out.println("ABCD".length(2));
                       ^
BadLength2.java:4: length() in java.lang.String cannot be applied to
(java.lang.String)
    System.out.println("ABCD".length("UV"));
                       ^
2 errors
```

Comme pour un appel de méthode de classe, le compilateur indique donc que la méthode `length` ne peut pas être utilisée avec des paramètres.

L'exécution d'un appel de méthode d'instance est complètement identique à celle d'une méthode de classe. Nous reviendrons en détail sur ce point au chapitre 9, quand nous apprendrons à créer nos propres types objet et donc nos propres méthodes d'instance.

Pourquoi des méthodes d'instance ?

Il n'existe pas vraiment de justification technique à l'utilisation de méthodes d'instance[6]. D'un point de vue pratique, on réduit simplement le texte à taper. Si, pour obtenir le nombre de caractères d'une chaîne, on devait utiliser une méthode de classe, on écrirait quelque chose comme `String.length(s)`, où `s` désigne un objet `String`. Il est plus rapide d'écrire `s.length()`.

[5] Le compilateur ne fait pas ici de distinction entre les méthodes de classe et les méthodes d'instance. Il indique simplement que la seule méthode `length` de la classe `String` ne peut pas prendre comme paramètre un objet `String`.

[6] Le langage ADA 95 manipule comme `Java` des objets et n'utilise pas la notion de méthode d'instance.

REMARQUE

Notons que l'appel `String.length(s)` **n'est pas possible**, comme le montre l'exemple 7.15.

Notation pour les méthodes d'instance

Pour documenter des méthodes d'instance, nous utiliserons les conventions de présentation proposées à la section 3.4.1. Pour la méthode `length`, nous obtenons la description qui suit.

La classe `String` définit la méthode d'instance suivante :

`int length()`

Cette méthode (sans paramètre) renvoie la **longueur** (le nombre de caractères) de l'objet appelant (qui représente une chaîne de caractères).

7.2.3 Manipulations au niveau du caractère

Motivation

Nous avons vu jusqu'à présent des applications essentiellement mathématiques de la programmation. Dans la pratique, on utilise un ordinateur pour bien d'autres applications, notamment le traitement de textes (au sens large du terme). Le prototype d'une opération de traitement de texte est par exemple le calcul de statistiques sur un texte : combien des lettres dans le texte, combien de mots, etc. Pour pouvoir travailler sur un texte, il faut impérativement accéder aux caractères qui composent ce texte.

Deux méthodes d'instance

Pour connaître le nombre de caractères d'une chaîne, on utilise la méthode d'instance `length` présentée dans la section précédente. Pour accéder à un caractère donné, on utilise la méthode d'instance de la classe `String` qui suit :

`char chatAt(int index)`

Cette méthode renvoie le caractère situé à la position `index` dans la chaîne de caractères appelante. On numérote les caractères de gauche à droite, en donnant au premier le numéro 0.

Commençons par un exemple simple d'utilisation des deux méthodes :

Exemple 7.17 :

On considère le programme suivant :

AfficheTexte

```
1  import dauphine.util.*;
2  public class AfficheTexte {
3    public static void main(String[] args) {
4      Console.start();
5      System.out.print("Entrez un mot : ");
6      String mot=Console.readString();
7      for(int index=0;index<mot.length();index++) {
8        System.out.println(mot.charAt(index));
9      }
10   }
11 }
```

Ce programme affiche chaque caractère du texte saisi sur une ligne distincte. Le point important à noter est que la boucle `for` va de 0 à `mot.length()-1` (inclus). En effet, comme les caractères sont numérotés à partir de 0, le dernier a pour numéro $n-1$ si la chaîne est longueur n. Voici un exemple d'affichage produit par le programme :

AFFICHAGE

```
Entrez un mot : bonjour
b
o
n
j
o
u
r
```

Problèmes d'accès

Si on tente d'accéder à un élément qui n'existe pas, le processeur produit une **exception**, qui provoque l'arrêt du programme. Avant de terminer son exécution, le programme affiche un message assez complexe qui comporte le texte `String index out of range`: suivi de la valeur numérique correspondant au caractère non existant. Si on tente `t.charAt(4)`, où `t` fait référence à une chaîne de caractères ne comportant pas de 5-ième caractère, on obtiendra donc le message `String index out of range`: 4. L'exemple suivant illustre cette situation.

Exemple 7.18 :

Voici un programme qui tente d'accéder à un caractère inexistant :

OutOfRange

```
1 import dauphine.util.*;
2 public class OutOfRange {
3   public static void main(String[] args) {
4     String toto="abcd";
5     System.out.println(toto.charAt(4));
6   }
7 }
```

Quand on lance ce programme, on obtient l'affichage suivant :

ERREUR D'EXÉCUTION

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 4
        at java.lang.String.charAt(String.java:507)
        at OutOfRange.main(OutOfRange.java:5)
```

Il faut bien noter qu'il s'agit d'une erreur **d'exécution**, pas de compilation.

Applications simples

Pour illustrer les possibilités des méthodes étudiées, donnons une solution élémentaire à un problème de statistique évoqué en introduction : combien trouve-t-on d'occurrences d'un caractère donné dans un texte ?

Voici un exemple de résolution de ce problème :

Exemple 7.19 :

Compte

```
1  import dauphine.util.*;
2  public class Compte {
3    public static int howManyChar(String texte,char lettre) {
4      int result=0;
5      for(int k=0;k<texte.length();k++) {
6        if (texte.charAt(k)==lettre) {
7          result++;
8        }
9      }
10     return result;
11   }
12   public static void main(String[] args) {
13     System.out.println("Dans \"Le soleil brille\", il y a "
14                        +howManyChar("Le soleil brille",'l')
15                        +" fois la lettre l");
16   }
17 }
```

Comment la méthode `howManyChar` fonctionne-t-elle ? Grâce à la boucle `for`, le processeur va étudier successivement toutes les lettres de la chaîne de caractères, dans l'ordre, de la première à la dernière. A chaque fois qu'il rencontre une lettre égale à la lettre recherchée, il incrémente de 1 la valeur de `result` qui contient le nombre de fois où la lettre voulue a été rencontrée. On peut donner l'algorithme de la méthode :

Données :
- `texte`, une chaîne de caractères
- `lettre`, un caractère

Résultat : le nombre d'occurrences de `lettre` dans `mot`.

1. placer 0 dans `result`
2. pour `k` allant de 0 à la longueur de `mot` diminuée de 1
   (a) si le caractère numéro `k` de `mot` est égal à `lettre` :
   ajouter 1 à `result`
3. Résultat : `result`

Voici maintenant un exemple de construction d'une chaîne résultat par l'intermédiaire d'une suite de concaténation :

Exemple 7.20 :

On souhaite "retourner" une chaîne de caractères, c'est-à-dire produire une nouvelle chaîne dont le contenu est celui de la chaîne de départ retourné. Voici une solution possible, implantée par la méthode `miroir` de la classe suivante :

Miroir

```
1  import dauphine.util.*;
2  public class Miroir {
3    public static String miroir(String s) {
4      String résultat="";
```

```
5       for(int i=0;i<s.length();i++) {
6         résultat=s.charAt(i)+résultat;
7       }
8       return résultat;
9     }
10    public static void main(String[] args) {
11      Console.start();
12      System.out.print("Entrez un texte : ");
13      String texte=Console.readString();
14      System.out.println(miroir(texte));
15    }
16  }
```

Voici un exemple d'affichage produit par le programme :

AFFICHAGE

```
Entrez un texte : Démonstration
noitartsnoméD
```

La méthode fonctionne très simplement : elle parcourt tous les caractères de la chaîne en les ajoutant à une chaîne résultat initialement vide. Comme l'ajout se fait à gauche, on obtient au final un texte inversé.

7.2.4 Retour sur la notion de classe

Nous savons depuis le chapitre 3 qu'une classe est un groupe d'éléments. Nous avons appris au chapitre 6 à définir des méthodes de classe qui sont l'un des éléments acceptables dans une classe. Le présent chapitre montre qu'une classe peut aussi contenir les éléments suivants :

- la description d'un type objet :
  pour pouvoir créer un objet, il faut définir la façon dont il va représenter ce qu'il est sensé représenter. Pour prendre un exemple simple, nous pouvons considérer `String` et `StringBuffer` (que nous étudierons à la section 7.4). Ces deux types objet représentent des chaînes de caractères. Il y a donc au moins deux façons différentes de représenter informatiquement un même concept. La classe `String` contient la description des éléments qui interviennent dans la représentation des chaînes de caractères choisie pour le type correspondant. Il en est de même pour la classe `StringBuffer` ;
- la programmation de méthodes d'instance :
  quand une classe définit un type objet, elle lui associe en général un ensemble de méthodes d'instance qui permettent une utilisation simple et efficace du type en question.

Nous étudierons comment définir ces nouveaux éléments au chapitre 9.

7.3 La gestion mémoire des objets

7.3.1 Le problème

Nous avons jusqu'à présent occulté discrètement un problème délicat posé par le type `String`. En effet, nous avons vu des types auxquels on pouvait associer un nombre de cases mémoire élémentaires (les types fondamentaux). Or, ce n'est évidemment pas le cas pour une chaîne de caractères : un texte de 3 lettres occupe moins de place dans la mémoire qu'un texte de 10000 lettres. Donc *a priori*, `String` ne peut pas vraiment être un type *classique*.

En fait, c'est plus précisément la notion de valeur qui pose problème. Une valeur **fondamentale**, c'est-à-dire d'un des types fondamentaux (cf. section 2.1.2, page 16), occupe un nombre de cases fixé (qui dépend de son type). Comme, d'après notre définition, une variable est un groupe de cases *fixé*, on peut utiliser les cases en question pour **écrire** la valeur (en fait sa représentation informatique).

Un objet de type `String` n'occupe pas un nombre de cases dépendant seulement de son type car une chaîne courte occupe moins de place qu'une chaîne longue, sans pour autant que l'une ou l'autre ne soit pas une chaîne de caractères. Un objet est donc représenté en mémoire par un groupe de cases dont le nombre **n'est pas fixé** par le type de l'objet. Comme nous l'avons expliqué à la section 7.2.4, ce type (c'est-à-dire en fait la classe associée) se contente en général de décrire la façon dont l'objet du monde "réel" est représenté en mémoire (voir le chapitre 9 pour des précisions).

Le principal problème est que pour des raisons techniques, une variable doit nécessairement occuper un nombre fixe de cases dans la mémoire. La solution à cette contradiction passe par la notion de **référence** et l'utilisation d'une séparation de la mémoire en deux morceaux : la **pile** et le **tas**.

7.3.2 Les références

Quel est donc le sens de la déclaration d'une variable de type `String` ? En fait, on indique que la variable déclarée est une **référence** sur l'objet qui va être manipulé. Une référence est un moyen informatique de repérer l'emplacement où est rangé un objet dans la mémoire. C'est en quelque sorte *l'adresse* de l'objet dans la mémoire. De plus, une référence prend toujours 4 cases (32 *bits*) de la mémoire[7], quelle que soit la taille occupée par l'objet qu'elle désigne. Le processeur permet d'utiliser une valeur particulière pour une référence : `null`. Cette valeur indique que la variable ne fait référence à aucun objet (cf la section 7.3.6 pour des précisions). Notons que comme pour les variables classiques, les références ne sont pas initialisées.

Comme une référence occupe un nombre de cases fixé, elle peut être rangée dans une variable, mais cela ne règle pas pour autant le problème des objets.

7.3.3 La pile et le tas

En fait, la mémoire de l'ordinateur est découpée en deux parties qui sont utilisées de façon différentes : la **pile** (*stack* en anglais) et le **tas** (*heap* en anglais).

La pile

Jusqu'à présent, nous avons exclusivement travaillé avec la pile. Cette zone de la mémoire possède deux propriétés importantes :

1. elle contient les variables déclarées dans le programme ;
2. le processeur peut créer des zones dans la pile afin de séparer la mémoire d'une méthode de celle d'une autre (voir la section 6.2.6).

Pour des raisons techniques complexes, ces propriétés imposent une contrainte : une partie du contenu de la pile doit nécessairement être déterminée à la compilation. En termes simples, pour que le compilateur soit capable de manipuler les variables grâce à leur nom, il faut impérativement que le contenu de chaque variable occupe un nombre de cases fixé **à la compilation**. C'est pourquoi les variables ne peuvent pas contenir directement les objets et qu'elles doivent être limitées aux types fondamentaux et aux références.

[7] pour la plupart des microprocesseurs actuels, à l'exception notable des DEC Alphas. Dans un futur proche, on passera à 8 cases (64 *bits*) sur tous les microprocesseurs, comme sur les Alphas, ce qui permet entre autre de gérer plus de mémoire.

Le tas

Le tas est la deuxième zone importante de la mémoire. Le tas est destiné à contenir les objets. Il n'est pas organisé en zone : le processeur ne peut pas placer de barrière dans le tas. De plus, il est impossible de manipuler **directement** le contenu du tas. Pour travailler avec un objet, il faut nécessairement connaître une référence vers cet objet. **La référence indique en fait l'emplacement de l'objet dans le tas**.

Dans la pile, la position des variables est déterminée à la compilation, ce qui permet aux programmes de les manipuler directement. Dans le tas, la position des objets est déterminée **dynamiquement**, ce qui oblige à utiliser des références. Par contre, les objets peuvent occuper un nombre de cases déterminé lui aussi **dynamiquement** c'est-à-dire à l'exécution du programme. En fait, les objets sont **créés** à l'exécution du programme, contrairement aux valeurs qui sont simplement écrites dans une zone mémoire dont la taille a été déterminée à la compilation.

REMARQUE

La division de la mémoire en pile et tas résulte donc d'un compromis : la pile permet une manipulation simple, mais n'autorise que les éléments dont la taille est déterminée à la compilation. Au contraire, la manipulation du tas est relativement complexe, mais il permet le stockage d'éléments dont la taille est déterminée à l'exécution. La manipulation par référence autorise aussi des manipulations impossibles à réaliser autrement (les effets de bord, cf les sections 7.4.6 et 7.4.7), ainsi qu'une efficacité supérieure dans certains cas (voir la section 7.3.5).

7.3.4 Représentation de la mémoire

La manipulation des références est parfois très délicate, c'est pourquoi il est utile de recourir à une représentation graphique de la mémoire. Nous avons déjà vu une telle représentation à la section 2.1.4 et nous l'avons complétée à la section 6.2.6. Nous devons maintenant introduire la distinction pile *versus* tas et proposer une représentation des références.

Étudions pour commencer le sens d'une déclaration et d'une affectation simple :

```
String texte;
texte="Bonjour";
```

La première ligne crée une variable de type `String` qui ne contient au départ aucune référence (et non pas la référence `null`). On ne peut donc pas afficher le contenu de `texte` qui n'est pas encore défini (aucune différence avec une variable d'un type fondamental).

La seconde ligne demande au processeur de réaliser deux opérations :

1. créer un objet de type `String` et le placer dans la mémoire. Cet objet représentera le texte `Bonjour` ;
2. placer dans la variable `texte` l'adresse de (ou la référence vers) l'objet qui vient d'être créé.

La figure 7.1 donne une représentation de l'état de la mémoire après l'exécution de l'affectation. Détaillons cette représentation :

- la mémoire est séparée en pile et tas, avec une identification claire de chaque partie ;
- les variables (ici la variable `texte` seule) sont représentées comme dans les sections 2.1.4 et 6.2.6, par une case précédée du nom de la variable et contenant son type ;
- dans le cas des types fondamentaux (non représenté dans cet exemple), la case contient la valeur éventuelle de la variable ;
- dans le cas des objets, la case de la variable est le point de départ d'une flèche qui représente la référence que la variable contient ;
- le point d'arrivée de la flèche est l'objet désigné par la référence : on représente un objet par une case, contenant le type de l'objet et sa "valeur".

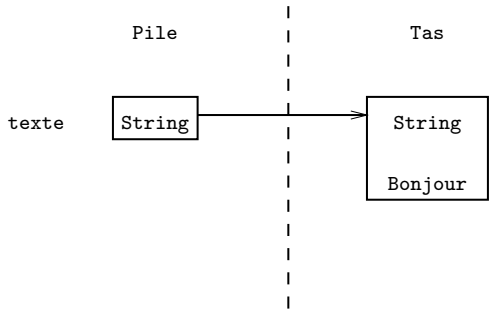

FIG. 7.1 – Création d'un objet et manipulation par référence

REMARQUE

Il ne s'agit en aucun cas de donner un ensemble de règles formelles pour représenter la mémoire (taille des cases, longueur des flèches, etc.) mais bien de permettre une représentation simplifiée pour analyser plus efficacement des programmes complexes. Nous respecterons donc les principes de représentation illustrés par l'exemple qui précède, en nous attachant avant tout à l'aspect pratique : le but est d'illustrer l'évolution de la mémoire, pas d'ajouter à la difficulté de compréhension une difficulté de représentation !

7.3.5 Retour sur l'affectation

L'affectation proprement dite

Dès le chapitre 2, nous avons étudié une des instructions les plus utilisées, l'affectation (voir la section 2.2). Rappelons l'interprétation par le processeur d'une affectation :

1. le processeur évalue l'expression située à droite du symbole d'affectation (`=`) ;
2. le processeur place la valeur obtenue dans la variable situé à gauche du symbole d'affectation.

Cette interprétation (la sémantique de l'affectation) n'est absolument pas remise en question par l'utilisation des références. Il faut cependant comprendre que les variables contiennent des références, pas des objets. Quand on place le contenu d'une variable dans une autre, on se contente donc de **recopier les références, pas les objets**.

Pour bien comprendre le fonctionnement de l'affectation, étudions un exemple très simple :

```
String texte1,texte2;
texte1="Bonjour";
texte2=texte1;
```

Voici comment interpréter ce programme :

1. la première ligne se contente de créer deux variables (dans la pile). Les variables ne contiennent pour l'instant aucune valeur ;
2. la deuxième ligne a les effets suivants :
   (a) le processeur crée un objet de type `String` (dans le tas) de valeur le texte `Bonjour` ;
   (b) le processeur place la référence vers cet objet dans la variable `texte1` ;
3. la troisième ligne est interprétée de la façon suivante :
   – le processeur évalue l'expression `texte1`. Il obtient comme valeur la référence vers l'objet représentant le texte `Bonjour`, créé à l'instruction précédente ;

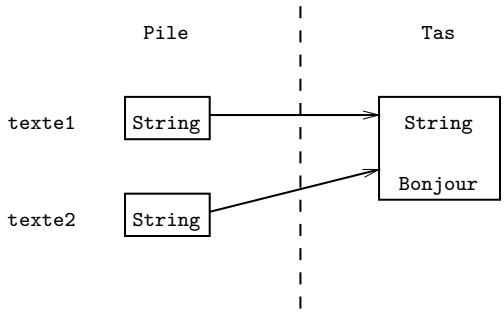

FIG. 7.2 – Deux variables font référence au même objet

– le processeur place la référence obtenue dans la variable `texte2`.

Après l'exécution de la troisième ligne, les deux variables **font donc référence au même objet**. La figure 7.2 représente la mémoire après l'exécution de l'affectation.

REMARQUE

La représentation proposée par la figure 7.2 utilise bien entendu deux flèches distinctes. La flèche n'est donc pas une représentation du *contenu* de la variable (les contenus des variables `texte1` et `texte2` sont strictement identiques), mais permet *d'illustrer le lien* entre la variable et l'objet : la variable *désigne* l'objet.

Il très important de comprendre que l'affectation ne recopie **jamais** l'objet référencé par l'opération. C'est un avantage et un inconvénient de la manipulation par référence :

– cela permet de conserver à l'affectation toute son **efficacité** : on ne recopie qu'une référence (quatre cases dans la mémoire), même si l'objet référencé est une chaîne très longue. L'opération est donc rapide et elle évite de plus le gaspillage de mémoire ;
– deux variables *distinctes* peuvent être utilisées pour manipuler un objet *unique*. Cela aura des conséquences difficiles à maîtriser (voir la section 7.4.6) ;
– lors d'un passage de paramètre (ou lors de la définition d'un résultat), seule la référence sera recopiée, pas l'objet : la méthode appelée et la méthode appelante pourront donc manipuler le *même* objet, ce qui aura des avantages et des inconvénients (voir la suite de la présente section ainsi que la section 7.4.7).

Dans les méthodes

Les mécanismes de passage de paramètres (section 6.3.3) et de définition d'un résultat (section 6.4.4) sont basés sur la copie des valeurs transmises. Nous venons de voir que la copie de valeur est toujours de mise pour les objets, mais qu'en fait seule les références sont copiées lors d'une affectation. C'est exactement la même chose lors d'une transmission d'information entre deux méthodes : **quand une méthode transmet un objet à une autre méthode, seule la référence est transmise : l'objet n'est pas copié**.

Pour bien comprendre les mécanismes, étudions un exemple volontairement complexe :

Exemple 7.21 :

On considère le programme suivant :

PassageReference

```
1 public class PassageReference {
2   public static String manipulation(String a,String b) {
3     a=b;
```

```
4     return b;
5   }
6   public static void main(String[] args) {
7     String u="TOTO",v="ABCD";
8     String w=manipulation(u,v);
9     System.out.println(u);
10    System.out.println(v);
11    System.out.println(w);
12  }
13 }
```

L'affichage produit par le programme est le suivant :

AFFICHAGE

```
TOTO
ABCD
ABCD
```

On voit que `u` n'est pas modifié par l'appel, ce qui est parfaitement normal : le contenu d'une variable d'une méthode ne peut pas être modifié par les instructions d'une autre méthode. De façon générale, l'affichage produit semble parfaitement normal. La grosse différence entre les types fondamentaux et les types objet réside dans l'interprétation de l'effet du programme sur la mémoire. Lors de l'appel de la méthode, par exemple, les `String`s ne sont pas recopiées : `a` désigne donc la même chaîne que `u`, par exemple. La figure 7.3 représente l'état de la mémoire juste avant l'exécution de la ligne 3, c'est-à-dire après le passage de paramètres.

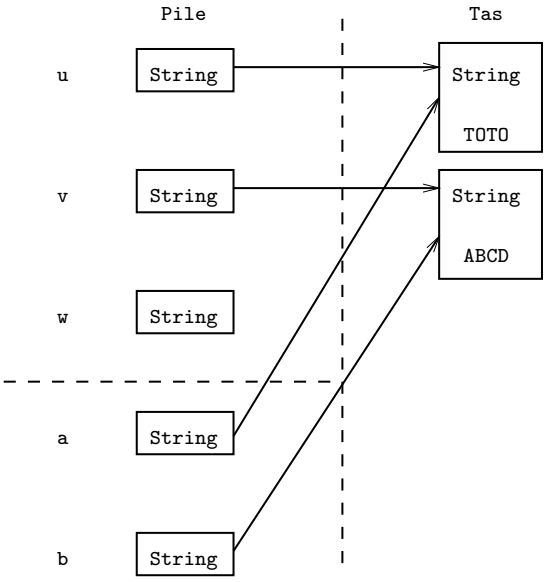

FIG. 7.3 – Passage de paramètres et références

Quand la ligne 3 est exécutée, on modifie simplement le contenu de la variable locale `a` (le paramètre formel), qui désigne ensuite le même objet que la variable `b` (la représentation de la chaîne "ABCD"). Le résultat de cette opération est illustrée par la figure 7.4

Quand la ligne 4 est exécutée, le processeur définit la valeur de retour de la méthode qui est la

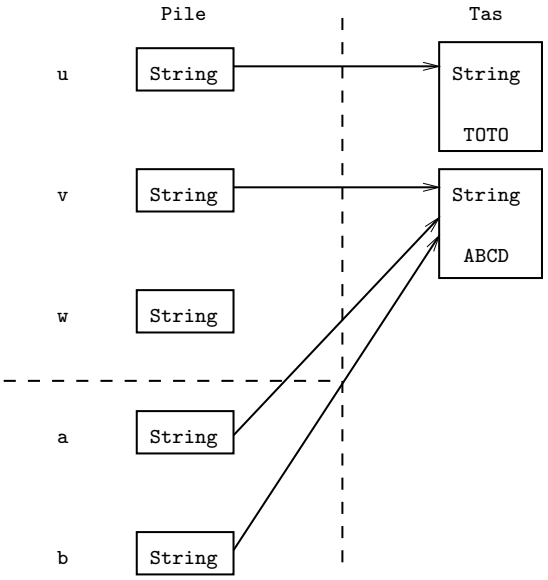

FIG. 7.4 – Copie d'une référence dans une variable locale

référence contenue dans `b`. La figure 7.5 représente l'état la mémoire au retour dans la méthode `main`.

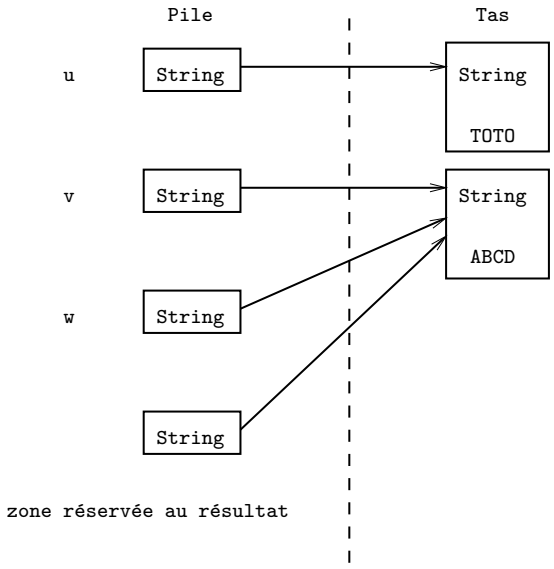

FIG. 7.5 – Référence comme résultat

Nous étudierons d'autres conséquences du mécanisme de passage de paramètres et de définition du résultat à la section 7.4.7.

7.3.6 La référence `null`

Nous savons depuis la section 2.2.5 qu'il est indispensable de donner une valeur à une variable avant de pouvoir l'utiliser. Cette règle s'applique bien entendu aux variables d'un type objet quel-

conque, comme l'illustre l'exemple suivant :

Exemple 7.22 :

Dans le programme suivant, on essaie d'utiliser une variable de type `String` sans lui avoir donné de contenu :

PasDeValeur

```
1 public class PasDeValeur {
2   public static void main(String[] args) {
3     String s;
4     System.out.println(s);
5   }
6 }
```

Le compilateur refuse le programme et donne le message d'erreur suivant :

Erreur de compilation

```
PasDeValeur.java:4: variable s might not have been initialized
    System.out.println(s);
                       ^
1 error
```

Le point nouveau pour les types objet est l'existence d'une référence particulière, la référence `null`. Cette valeur ne possède pas de type et elle peut donc être placée dans toute variable possédant un type objet (elle reste impossible à placer dans une variable d'un type fondamental). Sa sémantique est simple : elle indique que la variable qui la contient le désigne aucun objet. Considérons d'abord un exemple trompeur :

Exemple 7.23 :

Le programme suivant est parfaitement correct :

Null

```
1 public class Null {
2   public static void main(String[] args) {
3     String s=null;
4     System.out.println(s);
5   }
6 }
```

Il produit l'affichage suivant :

Affichage

```
null
```

A la lumière de cet exemple, on pourrait donc croire que la variable `s` contient une référence vers le texte "null". Il n'en est rien. La méthode `println` est tout simplement capable de faire la différence entre une référence vers une `String` (elle affiche alors la chaîne correspondante) et la référence `null` qui ne correspond à aucun objet (la méthode affiche alors le texte "null").

Pour bien comprendre l'interprétation de `null`, étudions l'exemple suivant :

Exemple 7.24 :

On considère le programme suivant :

PasDObjet

```
1 public class PasDObjet {
2   public static void main(String[] args) {
3     String s=null;
4     System.out.println(s.length());
5   }
6 }
```

Ce programme compile sans problème. Par contre, son exécution est interrompue par une erreur :

ERREUR D'EXÉCUTION

```
Exception in thread "main" java.lang.NullPointerException
        at PasDObjet.main(PasDObjet.java:4)
```

En effet, la référence `null` (*null pointer* en anglais) ne correspond à aucun objet. Donc, la tentative d'appel de la méthode d'instance `length` ne peut pas réussir, car une méthode d'instance nécessite un objet appelant. La figure 7.6 représente l'état de la mémoire juste avant l'exécution de la ligne 4 du programme. Cette illustration précise que la variable `s` contient la valeur `null` et ne correspond donc à aucun objet (d'où l'absence de flèche partant de `s`).

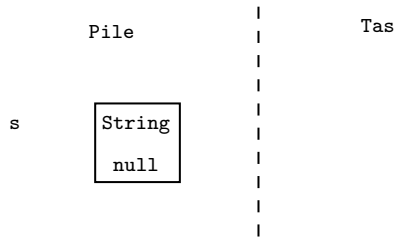

FIG. 7.6 – La variable `s` contient la référence `null`

On peut se demander pourquoi le compilateur accepte la construction et que l'erreur n'est détectée qu'à l'exécution du programme. La réponse est simple : les objets sont créés dynamiquement (en général par l'intermédiaire de constructeur, comme expliqué à la section 7.4.3). Il est donc en général impossible de savoir à la compilation (statiquement) si une variable va bien contenir une référence vers un objet (et non pas `null`) lors de l'exécution (dynamiquement). Le compilateur ne peut donc pas, en général, détecté ce genre de problème.

L'exemple suivant donne une des applications possibles de `null` et illustre l'impossibilité pour le compilateur de vérifier qu'une variable désigne bien un objet.

Exemple 7.25 :

On souhaite réaliser une méthode qui fabrique une sous-chaîne à partir d'une chaîne de caractères donnée. La sous-chaîne est constitué des caractères de la chaîne d'origine depuis le caractère de position `départ` (inclus) jusqu'à celui de position `fin` (non inclus). Si les paramètres `départ` et `fin` sont mal choisis, il est impossible de produire un résultat pertinent. C'est le cas par exemple si `départ` ne désigne pas une position correcte, ou encore si `fin` est plus petit que départ. La méthode `subString` qui fabrique la sous-chaîne commence par éliminer certains cas (lignes 4, 5 et 6). Pour ce faire, elle renvoie `null` au lieu d'une chaîne quelconque. De ce fait, la méthode ne renvoie un résultat qui correspond réellement à un objet `String` seulement dans le cas où ce résultat a un sens.

Voici le programme proposé :

SousChaine

```
1  import dauphine.util.*;
2  public class SousChaine {
3    public static String subString(String s,int départ,int fin) {
4      if( fin<départ || départ>=s.length() || départ<0) {
5        return null;
6      }
7      String résultat="";
8      fin=Math.min(fin,s.length());
9      for(int i=départ;i<fin;i++) {
10       résultat+=s.charAt(i);
11     }
12     return résultat;
13   }
14   public static void main(String[] args) {
15     Console.start();
16     System.out.print("Entrez un texte : ");
17     String texte=Console.readString();
18     System.out.print("Début du sous-texte : ");
19     int d=Console.readInt();
20     System.out.print("Fin du sous-texte : ");
21     int f=Console.readInt();
22     String sousTexte=subString(texte,d,f);
23     System.out.println("Résultat : "+sousTexte);
24   }
25 }
```

On remarque que la méthode `main` effectue des saisies. De ce fait, le compilateur ne peut pas savoir (au moment de la compilation du programme) ce que l'utilisateur va saisir à l'exécution. Il ne peut donc pas savoir si la méthode pourra faire un calcul pertinent et renvoyer une référence vers une `String`, ou si au contraire, le calcul sera impossible et que la méthode renverra la référence `null`. Le compilateur doit donc accepter le programme, le processeur se chargeant à l'exécution de découvrir d'éventuels problèmes.

Voici deux exemples d'interaction avec le programme :

Affichage

```
Entrez un texte : Le soleil brille
Début du sous-texte : 3
Fin du sous-texte : 9
Résultat : soleil
```

Affichage

```
Entrez un texte : Le soleil brille toujours
Début du sous-texte : 3
Fin du sous-texte : 2
Résultat : null
```

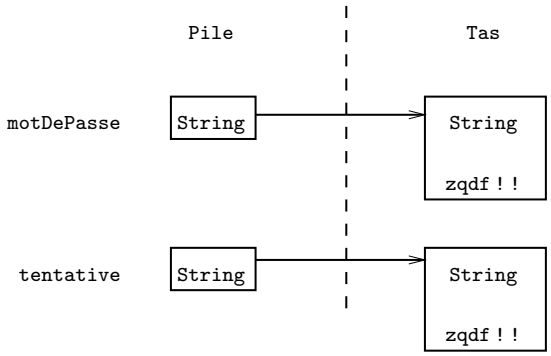

FIG. 7.7 – Deux objets distincts d'apparences identiques

7.3.7 Les comparaisons

Une première application importante des chaînes de caractères est l'interaction avec l'utilisateur. Jusqu'à présent, nous ne savions pas comment faire saisir à l'utilisateur autre chose qu'une valeur numérique. Grâce aux chaînes de caractères, il est possible de saisir des textes. Une première application simple est de vérifier si l'utilisateur a bien saisi une réponse autorisée (par exemple "oui" ou "non"). Pour ce faire, il faut pouvoir comparer entre elles deux chaînes de caractères. Or, l'utilisation de l'opérateur `==` réserve quelques surprises, comme l'illustre l'exemple suivant :

Exemple 7.26 :

On considère le programme suivante qui simule (de façon très naïve) une identification par mot de passe :

MotDePasseIncorrect

```
1  import dauphine.util.*;
2  public class MotDePasseIncorrect {
3    public static void main(String[] args) {
4      Console.start();
5      String motDePasse="zqdf!!";
6      System.out.print("Donnez le mot de passe : ");
7      String tentative=Console.readString();
8      if (tentative==motDePasse) {
9        System.out.println("Accès autorisé");
10     } else {
11       System.out.println("Accès refusé");
12     }
13   }
14 }
```

Contrairement à ce qu'on pourrait croire, ce programme affiche les lignes suivantes :

AFFICHAGE

```
Donnez le mot de passe : zqdf!!
Accès refusé
```

Pour comprendre cet exemple, il faut savoir interpréter la méthode `readString` et l'opérateur `==` :

– la méthode `readString` de la classe `Console` réalise une saisie. Quand l'utilisateur tape un texte, la méthode crée un **nouvel objet** de type `String` dans le tas et renvoie la référence vers

cet objet. La figure 7.7 représente l'état de la mémoire après l'exécution de la ligne 7, dans le cas où l'utilisateur saisit le texte `"zqdf ! !"`. On est ici dans la situation des **jumeaux**. Les deux objets `String` représentent la même chaîne de caractères. Ils semblent donc totalement identiques. Pourtant, ils sont distincts et correspondent donc à des **références différentes** (ils occupent des emplacements différents dans le tas) ;

– l'opérateur `==` effectue une comparaison du **contenu** des variables. Dans le cas d'objets, **il compare donc les références**. De ce fait, même si les objets représentent la même chaîne de caractères, ils correspondent à des références distinctes : les contenus des variables `motDePasse` et `tentative` sont donc différents et l'expression `tentative==motDePasse` vaut donc `false`, ce qui explique l'affichage obtenu.

Il est donc impossible en général de comparer deux chaînes de caractères en utilisant l'opérateur `==`. Cet opérateur est très utile, car il permet de comparer deux objets. Mais dans certains cas, il ne convient pas. Comment faire par exemple pour comparer les chaînes de caractères représentées par deux objets différents ?

Certains types objets, comme par exemple les `String`s, définissent une méthode d'instance `equals` qui autorise la comparaison des valeurs représentées par les objets.

La classe `String` définit par exemple la méthode d'instance suivante :

`boolean equals(Object o)`

Si l'objet paramètre `o` est de type `String`, la méthode compare la chaîne de caractères représentée par l'objet appelant avec celle représentée par `o`. Elle renvoie `true` si les chaînes sont identiques. Dans tous les autres cas (si les chaînes sont distinctes ou si `o` ne désigne pas un objet de type `String`), la méthode renvoie `false`.

REMARQUE

Le type `Object` est une sorte de *joker* pour tous les types objets. Son utilisation est assez délicate et, dans le présent ouvrage, nous nous contenterons de quelques mentions brèves. Le lecteur intéressé par des précisions pourra se reporter à [10].

Voici un exemple d'application de la méthode considérée :

Exemple 7.27 :

On reprend le principe de l'exemple 7.26, mais en tenant compte du fait qu'on ne doit pas comparer les objets référencés par `motDePasse` et `tentative`, mais les chaînes qu'ils représentent :

MotDePasse

```
1  import dauphine.util.*;
2  public class MotDePasse {
3    public static void main(String[] args) {
4      Console.start();
5      String motDePasse="zqdf!!";
6      System.out.print("Donnez le mot de passe : ");
7      String tentative=Console.readString();
8      if (tentative.equals(motDePasse)) {
9        System.out.println("Accès autorisé");
10     } else {
11       System.out.println("Accès refusé");
12     }
13   }
14 }
```

Cette version fonctionne parfaitement, grâce à la méthode `equals`.

REMARQUE

Il important de noter que seuls certains types objet proposent une méthode `equals` qui autorise une comparaison des valeurs représentées. Nous verrons à la section 7.4.8 que les autres types possèdent une méthode `equals` qui réalise une comparaison des références, exactement comme l'opérateur `==`.

Il est parfois utile de savoir si une variable d'un type objet quelconque contient effectivement une référence vers un objet ou bien la référence `null`. Pour ce faire, il suffit simplement de comparer le contenu de la variable avec `null`. Si la variable s'appelle `x`, on écrira donc simplement `x==null`, expression qui vaut `true` si et seulement si la variable `x` contient la référence spéciale `null`, c'est-à-dire ne désigne aucun objet.

7.3.8 Le nettoyage du tas

Considérons un programme très simple :

```
String u="abcd";
u="efgh";
```

L'objet de type `String` qui représente la chaîne "abcd" n'est plus référencé après l'exécution de la deuxième ligne. Plus précisément, il n'existe plus dans la mémoire de variable qui contient une référence vers cet objet. De ce fait, il n'existe plus aucun moyen d'utiliser cet objet. On peut alors se demander ce que le processeur va en faire.

En fait, `Java` est muni d'un mécanisme appelé le *garbage collector*, terme américain qui signifie **éboueur**[8]. Le rôle de cet éboueur est de ramasser les ordures, c'est-à-dire les objets qui ne sont plus utilisés. Ce mécanisme est régulièrement mis en marche par le processeur et assure que la mémoire n'est pas encombrée par des objets inutiles.

7.4 Chaînes de caractères modifiables : le type `StringBuffer`

7.4.1 Les `Strings` ne sont pas modifiables

L'opération de concaténation, la seule opération sur les `String`s qui engendre une chaîne de caractères différentes des chaînes de départ, ne modifie pas les objets `String` avec lesquels elle travaille. Pour nous en convaincre, analysons un exemple de programme très simple :

```
String u="a",v="b",w;
w=u+v;
u+="bcde";
```

La ligne 2 réalise la concaténation des chaînes désignées par les variables `u` et `v`. Cette opération **fabrique un nouvel objet** de type `String` qui représente le résultat de la concaténation. La figure 7.8 donne l'état de la mémoire après la deuxième ligne.

La ligne 3 est plus intéressante. On pourrait croire en effet qu'elle modifie la chaîne de caractères représentée par l'objet `String` auquel `u` fait référence. Ce n'est pas du tout le cas. En fait, l'instruction `u+="bcde" ;` est interprétée comme `u=u+"bcde" ;`. L'évaluation de l'expression provoque la création d'un nouvel objet `String` qui représente le résultat de la concaténation. De ce fait, l'objet que désignait `u` avant l'affectation reste inchangé. La figure 7.9 représente l'état de la mémoire après l'exécution de la ligne 3.

[8] en français, il est assez commun d'utiliser une traduction un peu "niaise" pour le dispositif en question, à savoir ramasse-miettes, ou, pire encore, glaneur de cellules.

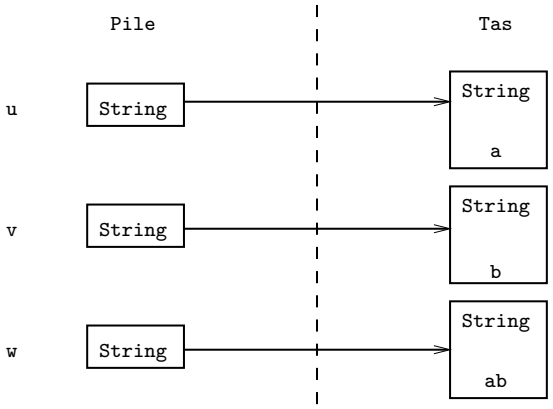

FIG. 7.8 – Résultat d'une concaténation simple (`w=u+v ;`)

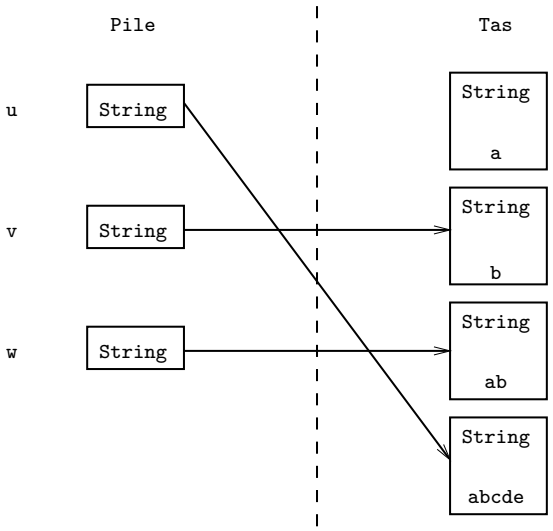

FIG. 7.9 – Résultat d'une concaténation (`u+="bcde" ;`)

REMARQUE

Nous n'indiquons ici que le cas de l'opérateur de concaténation, mai il faut noter qu'il est **strictement impossible de modifier la chaîne de caractères représentée par un objet de type** `String`.

7.4.2 Problème d'efficacité

Comme nous venons de le voir, les objets `String` ne sont pas modifiables, et se comportent donc comme des valeurs classiques (c'est-à-dire d'un type fondamental).

Ceci a un inconvénient majeur que nous allons évoquer grâce à l'exemple qui suit :

Concat

```
1 public class Concat {
2   public static void main(String[] args) {
3     String message;
4     message="a"+"b"+"c"+"d";
5     System.out.println(message);
6   }
7 }
```

On se contente ici de mettre bout à bout quatre chaînes réduite chacune à un caractère, afin de fabriquer une dernière chaîne qui est affichée. Il est intéressant de comprendre comment l'opération est effectuée. Comme pour tout calcul, l'ordinateur est simplement capable "d'additionner" deux chaînes. Donc, l'addition est réécrite[9] avec des parenthèses en :

```
message=((("a"+"b")+"c")+"d");
```

Quelle est la conséquence de ces trois additions sur la mémoire de l'ordinateur ? Tout d'abord, le processeur doit créer quatre objets `String` correspondant chacun à une des quatre chaînes de départ. Il effectue alors la première addition : il crée un objet `String` correspondant au texte `ab`. La deuxième addition se solde par la création d'un sixième objet `String` correspondant au texte `abc` et pour finir, un septième objet `String` est créé, correspondant au texte `abcd` et la variable `message` reçoit une référence sur cet objet. Dans cette opération, le processeur a donc créé deux objets inutiles, correspondant aux textes intermédiaires `ab` et `abc`. Or, la création de chaque objet implique la recopie des textes en question depuis les objets d'origine vers le nouvel objet : ici, on copie 5 lettres pour rien. Bien sûr, ce chiffre est faible, mais ceci est une simple conséquence des textes d'origine qui sont tous réduits à une seule lettre. On voit bien qu'avec des textes plus longs, on recopierait de nombreuses lettres de façon inutile. Les `String`s ne permettent donc pas une solution efficace, à la fois en terme de temps nécessaire au calcul, mais aussi en terme de mémoire occupée.

Quelle solution proposer ? Il faudrait pouvoir modifier un objet de type `String`. En effet, on commencerait alors par créer un tel objet correspondant au texte `ab` (2 copies de lettres), puis on ajouterait la lettre `c` (une copie) et la lettre `d` (une copie), soit un total de quatre copies. Dans la version précédente, le total est de cinq plus la dernière concaténation qui provoque quatre copies : neuf copies en tout.

Fort heureusement, il existe une classe, `StringBuffer`, qui propose des chaînes de caractères **modifiables**. Nous allons voir dans la suite du texte comment utiliser cette classe. La principale conséquence de son existence est la possibilité d'avoir des concaténations (et d'autres opérations) plus efficaces. Mais, son utilisation pose des problèmes intéressants qui font que cette classe ne peut pas remplacer la classe `String` mais seulement la compléter.

[9] En fait, elle *serait* réécrite de cette façon si les `StringBuffer` n'existaient pas.

REMARQUE

Nous sommes confronté de nouveau à la différence entre un concept et sa représentation informatique. En `Java`, il existe plusieurs types fondamentaux pour représenter un entier relatif (`int` et `long` par exemple). De la même façon, il existe plusieurs types objet pour représenter une chaîne de caractères : `String` et `StringBuffer`.

7.4.3 Le type `StringBuffer` et les constructeurs associés

`StringBuffer`

Nous allons donc utiliser une nouvelle classe : `StringBuffer`. Le premier problème qui se pose est celui de la création d'un objet instance de la classe `StringBuffer`. En effet, comme nous l'avons indiqué à la section 7.1.5, seul le type objet `String` possède des valeurs littérales. De plus, les types `String` et `StringBuffer` ne sont pas directement compatibles, comme le montre l'exemple suivant :

Exemple 7.28 :

On tente naïvement d'utilisateur une valeur littérale de type `String` pour fabriquer un objet de type `StringBuffer` :

BadString

```
1 public class BadString {
2   public static void main(String[] args) {
3     StringBuffer message="Bonjour";
4   }
5 }
```

Le compilateur refuse le programme et affiche le message suivant :

ERREUR DE COMPILATION

```
BadString.java:3: incompatible types
found   : java.lang.String
required: java.lang.StringBuffer
    StringBuffer message="Bonjour";
                         ^
1 error
```

Le compilateur indique donc clairement que les deux types `String` et `StringBuffer` ne sont pas compatibles.

Les constructeurs

Pour tous les types objet excepté `String`, le seul moyen de créer un objet est de passer par un **constructeur**. Un **constructeur** est une méthode spéciale d'une classe qui se charge de créer un objet du type correspondant à la classe. Le nom de cette méthode est toujours le nom de la classe. Une classe peut posséder plusieurs constructeurs à condition que chaque constructeur demande des paramètres différents (c'est la situation normale des méthodes, voir la remarque 3.4.2). L'utilisation générale d'un constructeur est :

Nom du type de la variable *identificateur*=new *Nom de la Classe*(*paramètres*);

L'instruction `new` indique qu'on demande au processeur de créer un objet. Le nom du type de la variable est aussi le nom de la classe. Le constructeur portant nécessairement le nom de la classe, l'expression qui suit le symbole d'affectation `=` correspond à l'utilisation du constructeur. Dans certains cas, on utilise un constructeur sans paramètre, mais les parenthèses restent obligatoires. Dans la pratique, les constructeurs jouent pour les objets le rôle joué par les valeurs littérales pour les types fondamentaux.

Pour la classe `StringBuffer`, on utilisera avant tout les constructeurs suivants :

- `StringBuffer()`.
  Ce constructeur sans paramètre fabrique un objet `StringBuffer` correspondant à la chaîne de caractères vide.
- `StringBuffer(String s)`.
  Ce constructeur crée un objet `StringBuffer` correspondant à la chaîne de caractères représentée par `s`.

Exemple 7.29 :

Voici un exemple simple d'utilisation du second constructeur :

Create

```
1 public class Create {
2   public static void main(String[] args) {
3     StringBuffer message=new StringBuffer("Bonjour");
4     System.out.println(message);
5   }
6 }
```

Comme on peut le deviner, ce programme affiche `Bonjour`.

REMARQUE

On remarque dans l'exemple précédent qu'on peut afficher directement un objet de classe `StringBuffer`. En fait, de façon générale, on peut tout afficher avec `System.out.println`, le processeur se chargeant de traduire "automatiquement" le paramètre de la méthode en une chaîne de caractères. Nous reviendrons sur ce point au chapitre 9.

Le cas de `String`

Il faut noter que même si il est pratique d'utiliser les valeurs littérales de type `String`, la classe correspondante définit un ensemble de constructeurs dont voici un sous-ensemble utile :

- `String()`.
  Ce constructeur sans paramètre fabrique un objet `String` correspondant à la chaîne de caractères vide.
- `String(String s)`.
  Ce constructeur crée un objet `String` correspondant à la chaîne de caractères représentée par `s`. Les deux objets sont totalement indépendants.
- `String(StringBuffer s)`.
  Ce constructeur crée un objet `String` représentant la même chaîne de caractères que l'objet `s` paramètre. Les deux objets sont totalement indépendants et une modification du `StringBuffer` n'aura aucune influence sur l'objet `String` ainsi créé.

REMARQUE

Il faut noter que l'utilisation d'un constructeur a deux effets. Le constructeur fabrique un objet dans le tas, et l'expression de création (de la forme `new` *Nom de la Classe*`(`*paramètres*`)`) prend pour valeur la référence vers l'objet nouvellement créé. On peut donc utiliser cette expression partout où une référence vers un objet est attendue. On peut donc écrire : `int x=(new StringBuffer("ABCD")).length() ;`

L'exemple proposé n'est bien sûr par très pertinent, mais, dans la pratique, il est parfois utile de pouvoir créer un objet pour l'utiliser directement, sans placer sa référence dans une variable. Cela peut être le cas par exemple quand on doit utiliser un objet comme paramètre d'une méthode.

7.4.4 Modification d'un objet de type `StringBuffer`

Avant de présenter les méthodes d'instance qui rendent le type `StringBuffer` utile, commençons par une bonne nouvelle : les méthodes d'instance `length` et `charAt` de la classe `String` existent aussi pour la classe `StringBuffer` et se comportent de la même façon. Notons que cette situation est exceptionnelle : il est très rare que deux classes différentes définissent des méthodes de même nom et avec le même comportement.

Étudions maintenant des méthodes d'instance qui permettent de **modifier** un objet de type `StringBuffer`, car ce sont elles qui font la différence avec les `String` :

`void setCharAt(int n,char c)`

Cette méthode permet de changer un caractère dans la chaîne. Si par exemple `message` est un `StringBuffer`, l'appel `message.setCharAt(0,'A')` remplace la première lettre de la chaîne par un `A`.

`StringBuffer append(type x)`

Cet ensemble de méthodes permet d'ajouter la représentation de `x` en chaîne de caractères à la fin de l'objet `StringBuffer` appelant.

REMARQUE

La méthode `setCharAt` ne permet pas de créer des caractères. Si on dispose par exemple d'une variable `sb` faisant référence à un `StringBuffer` représentant le texte `"tot"`, on ne peut pas ajouter une lettre à ce texte en faisant `sb.setCharAt(3,'o')`. Cet appel de méthode provoque au contraire une erreur car on tente de modifier un caractère qui n'existe pas. Voir l'exemple 7.30 pour une illustration du problème.

Exemple 7.30 :

On considère la tentative de création d'un caractère proposée dans la remarque précédente :

BadSetCharAt

```
1 public class BadSetCharAt {
2   public static void main(String[] args) {
3     StringBuffer sb=new StringBuffer("tot");
4     sb.setCharAt(3,'o');
5     System.out.println(sb);
6   }
7 }
```

Le compilateur accepte sans problème le programme. Par contre, une erreur d'exécution se produit, ce qui engendre l'affichage suivant :

Erreur d'exécution

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 3
        at java.lang.StringBuffer.setCharAt(StringBuffer.java:362)
        at BadSetCharAt.main(BadSetCharAt.java:4)
```

Voici maintenant des exemples d'application des méthodes de modification :

Exemple 7.31 :

Reprenons le programme simple de concaténation proposé dans la section 7.4.2. Avec un `StringBuffer`, on écrit :

Concat2

```
1  public class Concat2 {
2    public static void main(String[] args) {
3      StringBuffer message=new StringBuffer();
4      message.append("a");
5      message.append("b");
6      message.append("c");
7      message.append("d");
8      System.out.println(message);
9    }
10 }
```

Le principe est donc simple : on ajoute peu à peu des éléments à la fin de l'objet désigné par la variable `message`. Comme cet objet est modifiable, il évolue afin de contenir à la fin du programme une représentation de la chaîne de caractères `"abcd"`. Les figures 7.10 et 7.11 illustrent l'évolution de la mémoire : le processeur ne crée pas d'objets intermédiaires correspondant aux chaînes `"ab"` et `"abc"` alors que c'était le cas avec les `String`. L'utilisation des `StringBuffer`s permet donc de rendre certains programmes plus efficaces.

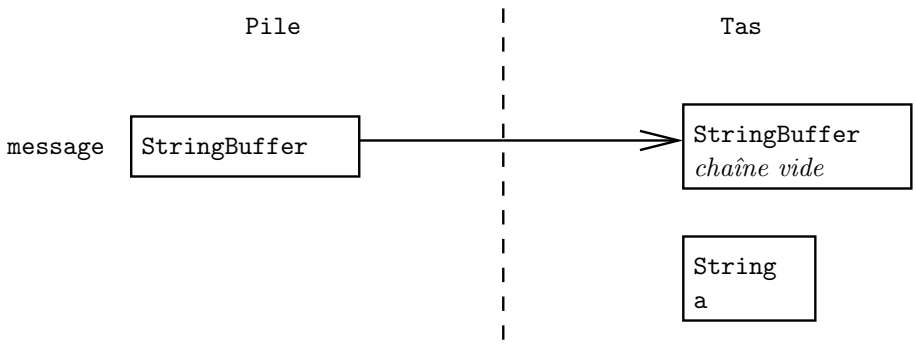

Fig. 7.10 – Mémoire dans le programme de l'exemple 7.31 **avant le premier** `append`

Exemple 7.32 :

Le programme suivant est un jeu du pendu simplifié :

Pendu

```
1  import dauphine.util.*;
2  public class Pendu {
3    public static int howManyChar(StringBuffer texte,char lettre) {
```

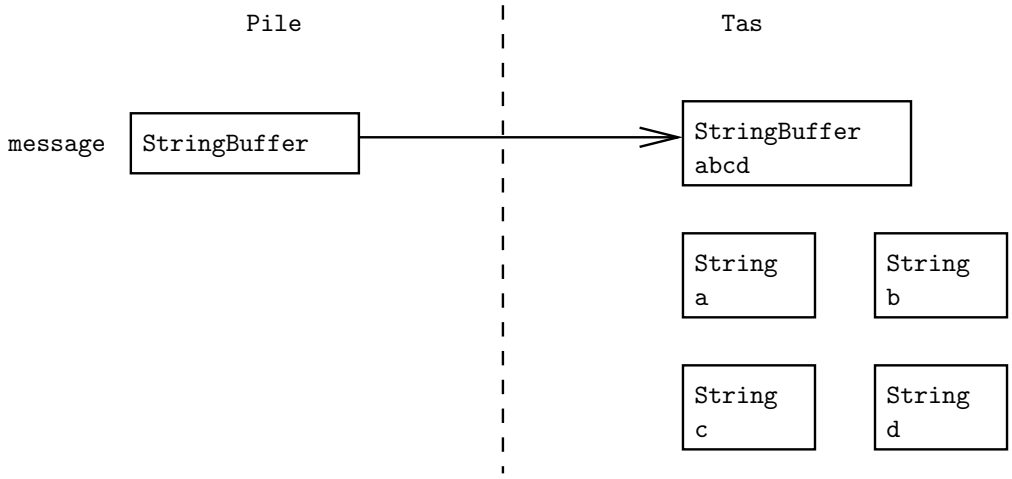

FIG. 7.11 – Mémoire dans le programme de l'exemple 7.31 **après le dernier** `append`

```
4     int result=0;
5     for(int k=0;k<texte.length();k++)
6       if (texte.charAt(k)==lettre)
7         result++;
8     return result;
9   }
10  public static StringBuffer toGuess(String texte) {
11    StringBuffer result=new StringBuffer(texte);
12    for(int i=0;i<result.length();i++)
13      result.setCharAt(i,'?');
14    return result;
15  }
16  public static void main(String[] args) {
17    Console.start();
18    String aDeviner="Bonjour";
19    StringBuffer mystere=toGuess(aDeviner);
20    char lettre;
21    while (howManyChar(mystere,'?')>0) {
22      System.out.println("Mot : "+mystere);
23      lettre=Console.readChar();
24      for(int i=0;i<mystere.length();i++)
25        if (aDeviner.charAt(i)==lettre)
26          mystere.setCharAt(i,lettre);
27    }
28    System.out.println("Mot : "+mystere);
29    System.out.println("Bravo !");
30  }
31 }
```

Expliquons les méthodes une à une :

– `howManyChar`
  Nous avons déjà vu cette méthode dans l'exemple 7.19. La seule différence est qu'elle utilise

ici un objet de la classe `StringBuffer` au lieu d'un objet de la classe `String`. Ceci ne change rien au corps de la méthode qui se base sur les méthodes d'instance `length` et `charAt` qui existent dans les deux classes. Cette méthode indique combien de fois un caractère apparaît dans une chaîne.

- `toGuess`
  Cette méthode fabrique un `StringBuffer` qui représente la chaîne qui lui est transmise en paramètre. La chaîne du `StringBuffer` est aussi longue que la chaîne de départ, mais chaque lettre est remplacée par un point d'interrogation pour indiquer que la lettre n'est pas connue. La méthode utilisée pour fabriquer le `StringBuffer` est très simple : on commence par le créer avec pour valeur initiale le texte paramètre de la méthode. L'objet créé a donc la bonne taille. Ensuite, on transforme chaque caractère grâce à une boucle et à la méthode `setCharAt`.
- `main`
  La méthode principale comporte essentiellement une boucle qui affiche le `StringBuffer` produit par `toGuess`. A chaque tour de la boucle, on demande à l'utilisateur une lettre grâce à `readCharAskAgain`. Ensuite, grâce à une autre boucle, on compare la lettre proposée avec toutes les lettres du mot à deviner. A chaque fois que la lettre proposée est égale à celle du mot (ce qui peut bien sûr arriver plus d'une fois), on remplace dans le `StringBuffer` le point d'interrogation par la lettre devinée. La boucle s'arrête quand le `StringBuffer` ne contient plus de point d'interrogation, c'est-à-dire quand l'utilisateur a trouvé toutes les lettres.

REMARQUE

Les méthodes `append` de la classe `StringBuffer` renvoient toutes une référence sur le `StringBuffer` appelant. Si on a deux variables `a` et `b` de type `StringBuffer`, écrire `a.append("toto") ;` suivi de `b=a` est strictement équivalent à écrire `b=a.append("toto") ;`. Ceci permet d'écrire par exemple `a.append("toto").append(" et titi") ;`, ce qui va bien coller successivement à la fin de la première chaîne représentée par `a` la chaîne `toto` puis la chaîne `  et titi`.

7.4.5 Conversions

Nous avons vu à la section 7.1.4 qu'il était possible de convertir n'importe quelle valeur d'un type fondamental en une `String`, en utilisant les méthodes `valueOf` de la classe `String`. Il n'existe pas de méthodes équivalentes pour les `StringBuffer`s. De ce fait, pour convertir un entier en `StringBuffer`, on doit passer par une `String` intermédiaire et écrire par exemple :

```
int x=12;
StringBuffer s=new StringBuffer(String.valueOf(x));
```

Bien entendu, la conversion directe est impossible, comme pour les `String`s. L'affectation

```
StringBuffer s=12;
```

est donc incorrecte et rejeté par le compilateur.

Il est possible de "convertir" un `StringBuffer` en une `String`. Pour ce faire, il faut utiliser la méthode d'instance suivante :

`String toString()`
  Cette méthode (sans paramètre) renvoie une représentation sous forme de `String` de l'objet appelant (un `StringBuffer`).

On peut donc écrire les lignes suivantes :

```
StringBuffer s=new StringBuffer("abc");
String t=s.toString();
```

La chaîne de caractères obtenue est un objet **distinct et indépendant** de l'objet `StringBuffer` de départ. Il faut noter qu'avec le constructeur adapté, cette méthode de conversion est la seule technique permettant de passer de `StringBuffer` à `String`. L'exemple suivant illustre les "conversions" rejetées par le compilateur :

Exemple 7.33 :

On considère le programme suivant :

BadStringBuffer

```
1 public class BadStringBuffer {
2   public static void main(String[] args) {
3     StringBuffer s=new StringBuffer("abc");
4     String t=s;
5     String u=(String)s;
6   }
7 }
```

Le programme est rejeté par le compilateur qui affiche les messages d'erreur suivants :

ERREUR DE COMPILATION

```
BadStringBuffer.java:4: incompatible types
found   : java.lang.StringBuffer
required: java.lang.String
    String t=s;
             ^
BadStringBuffer.java:5: inconvertible types
found   : java.lang.StringBuffer
required: java.lang.String
    String u=(String)s;
                     ^
2 errors
```

Les types `String` et `StringBuffer` sont jugés incompatibles.

REMARQUE

La méthode de conversion `toString` est centrale en `Java` et nous l'utiliserons pour tous les objets. Nous étudierons cette méthode plus précisément au chapitre 9.

La conversion d'un `StringBuffer` en une `String` associé aux possibilités de modification des premiers permet de proposer des versions efficaces du calcul du miroir d'une chaîne de caractères (exemple 7.20) :

Exemple 7.34 :

On propose donc le programme suivant :

MiroirStringBuffer

```
1 import dauphine.util.*;
2 public class MiroirStringBuffer {
3   public static String miroir1(String s) {
4     StringBuffer résultat=new StringBuffer();
```

```
5       for(int i=s.length()-1;i>=0;i--) {
6         résultat.append(s.charAt(i));
7       }
8       return résultat.toString();
9     }
10    public static String miroir2(String s) {
11      StringBuffer résultat=new StringBuffer(s);
12      for(int i=0;i<s.length();i++) {
13        résultat.setCharAt(i,s.charAt(s.length()-i-1));
14      }
15      return résultat.toString();
16    }
17    public static void main(String[] args) {
18      Console.start();
19      System.out.print("Entrez un texte : ");
20      String texte=Console.readString();
21      System.out.println(miroir1(texte));
22      System.out.println(miroir2(texte));
23    }
24  }
```

La méthode `miroir1` procède de façon très similaire à la méthode `miroir` de l'exemple 7.20 : le résultat est construit progressivement en ajoutant peu à peu les caractères souhaités à la fin de la chaîne résultat. Ici, on parcourt la chaîne de départ de la fin vers le début et on ajoute les caractères à la fin de la chaîne résultat. On obtient bien ainsi la chaîne de départ à l'envers. La solution est efficace parce qu'on peut modifier le `StringBuffer`.

La méthode `miroir2` utilise la technique de la modification des caractères plus que celle de l'ajout. Dans certaines circonstances, elle peut être plus efficace que la méthode `miroir1`, mais les raisons techniques de cette différence d'efficacité dépassent le cadre de ce chapitre. Disons pour simplifier que la modification de la longueur d'un `StringBuffer` prend en général plus de temps que la modification des caractères (sans ajout, ni suppression).

Voici un exemple d'affichage produit par le programme :

AFFICHAGE

```
Entrez un texte : Bozo le clown
nwolc el ozoB
nwolc el ozoB
```

7.4.6 Manipulation par référence et objets modifiables

Pour comprendre les conséquences de la manipulation par référence d'objets modifiable, étudions l'exemple suivant :

Exemple 7.35 :

On considère le programme suivant :

Echange

```
1   public class Echange {
2     public static void main(String[] args) {
3       String a="toto",b="tata",c;
```

```
4       c=a;
5       a=a+b;
6       StringBuffer d=new StringBuffer("toto"),e=new StringBuffer("tata"),f;
7       f=d;
8       d.append(e);
9       System.out.println(a+","+b+","+c);
10      System.out.println(d+","+e+","+f);
11    }
12  }
```

L'affichage produit est :

AFFICHAGE

```
tototata,tata,toto
tototata,tata,tototata
```

Le premier affichage est tout à fait classique. Il signifie qu'après les opérations, `a` désigne un objet qui représente la chaîne `tototata`, `b` désigne la chaîne `tata` alors que `c` désigne la chaîne `toto`.

Le second affichage peut paraître plus étrange. Il signifie en effet qu'après les opérations, `d` contient la chaîne `tototata`, `e` contient la chaîne `tata` alors que `f` contient la chaîne `tototata`. Pourtant, aucune opération n'est intervenue sur `f` après `f=d ;`. Pour comprendre ce qui s'est passé, il faut se rappeler que l'affectation `f=d ;` indique seulement que `f` désigne maintenant le même objet que `d`. Donc, l'opération `d.append(e)`, qui *modifie* l'objet auquel `d` fait référence, a un **effet de bord**, c'est-à-dire qu'elle semble modifier une variable qui n'apparaît pas explicitement dans le texte de l'opération. C'est parfaitement normal car l'objet modifié par l'appel de la méthode d'instance est celui auquel `f` fait *aussi* référence. La figure 7.12 donne le schéma de la mémoire pour les variables `d`, `e` et `f`. Pour les `String`, le problème ne se pose

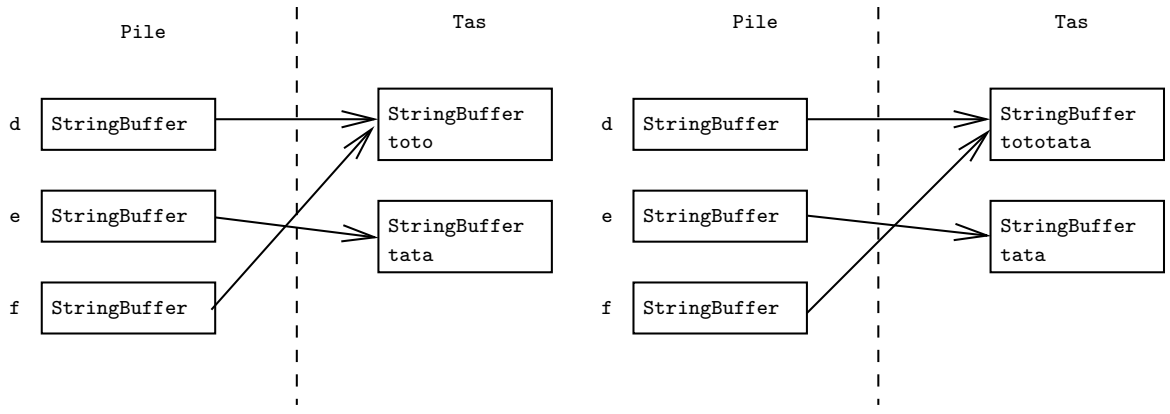

FIG. 7.12 – Modification d'un `StringBuffer`

pas car ces objets ne sont pas modifiables (cf la section 7.4.1). Donc, quand on écrit `a=a+b ;`, on indique simplement que le processeur doit fabriquer un nouvel objet obtenu en concaténant les objets auxquels `a` et `b` font référence, puis placer une référence sur cet objet dans `a`. La figure 7.13 donne le schéma de la mémoire pour les variables `a`, `b` et `c`.

Il est important de comprendre que la différence de comportement entre les `String` et les `StringBuffer` est la conséquence du caractère **modifiable** des objets la classe `StringBuffer` et du carac-

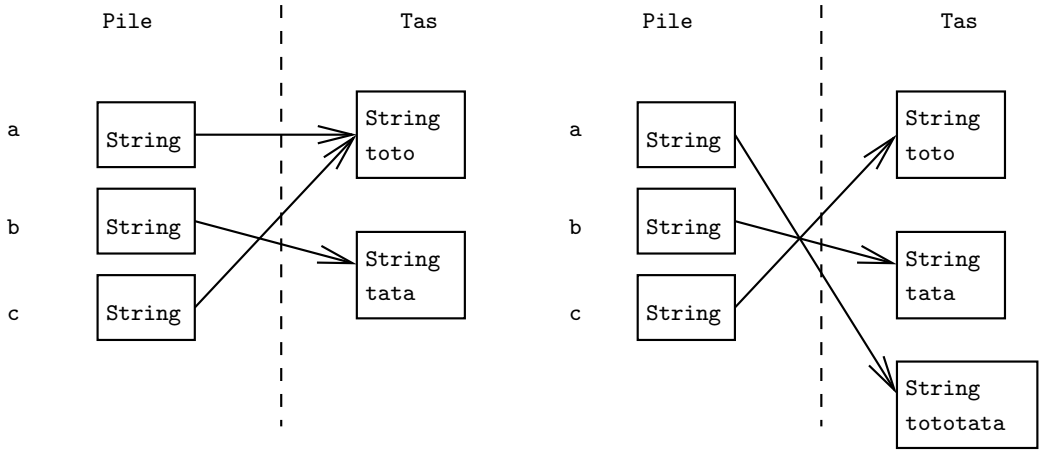

FIG. 7.13 – Les `String` ne sont pas modifiables

tère **immuable** de ceux de la classe `String`. Tous les objets dont la classe propose des méthodes de modification se comporteront comme les `StringBuffer` et tous les objets dont la classe assure l'impossibilité de modification se comporteront comme les `String` (c'est-à-dire approximativement comme les valeurs des types fondamentaux).

La raison en est simple : quand une variable `a` fait référence à un objet immuable, le seul moyen de changer sa valeur apparente est de lui faire désigner un *autre* objet. De ce fait, même si d'autres variables faisaient référence au même objet que `a`, aucune opération portant sur `a` n'a d'influence sur elles car le seul point commun entre ces variables et `a` est l'objet référencé qui est, par définition, immuable.

Par contre, quand deux variables font référence au *même* objet *modifiable*, on peut utiliser n'importe laquelle des deux pour modifier l'objet, ce qui a des conséquences sur la valeur *apparente* de l'autre variable. Une telle modification est un **effet de bord**.

Il ne faut cependant par croire que les effets de bord peuvent être magiques. **Il reste impossible de modifier le contenu d'une variable sans faire apparaître son identificateur à gauche du symbole d'affectation =**. La nouveauté provient du fait que le contenu non modifiable est simplement une **référence** dans le cas des objets. Quand l'objet lui-même est modifiable, on peut observer un effet de bord. Par contre, il est impossible de changer d'objet sans une utilisation directe de la variable. L'exemple suivant illustre ce point :

Exemple 7.36 :

On considère le programme suivant :

PasDEffetDeBord

```
1  public class PasDEffetDeBord {
2    public static void main(String[] args) {
3      StringBuffer a=new StringBuffer("TOTO");
4      StringBuffer b=a;
5      b.append('!');
6      System.out.println(a);
7      b=new StringBuffer("titi");
8      System.out.println(a);
9      System.out.println(b);
10   }
11 }
```

L'affectation de la ligne 4 indique que `a` et `b` font références au même objet. La ligne 5 provoque de ce fait un effet de bord, qu'on observe depuis `a` grâce à la ligne 6. Par contre, la ligne 7 se contente de changer le **contenu** de `b`, en lui donnant pour valeur une référence vers un **nouvel** objet. Le contenu de `a` ne peut pas être changé par cette opération. L'affichage total obtenu est donc le suivant :

Affichage

```
TOTO!
TOTO!
titi
```

7.4.7 Méthodes de classe et références

Les effets de bord apparaissent encore plus directement quand on utilise des méthodes, comme l'illustre le programme suivant :

Bord

```
1  public class Bord {
2    public static void changeDouble(double a) {
3      a=a+5.5;
4    }
5    public static void changeString(String a) {
6      a=a+" et tata";
7    }
8    public static void changeStringBuffer(StringBuffer a) {
9      a.append(" et tata");
10   }
11   public static void main(String[] args) {
12     double x=0.4;
13     String s="toto";
14     StringBuffer sb=new StringBuffer("toto");
15     changeDouble(x);
16     changeString(s);
17     changeStringBuffer(sb);
18     System.out.println(x);
19     System.out.println(s);
20     System.out.println(sb);
21   }
22 }
```

L'affichage produit est le suivant :

Affichage

```
0.4
toto
toto et tata
```

La première ligne n'est pas une surprise. En effet, on sait que la modification des paramètres d'une méthode n'a pas de conséquence dans la méthode qui l'appelle car les paramètres sont des variables locales de la méthode appelée dans lesquelles on recopie la valeur des paramètres effectifs (revoir

la section 6.3.3). Donc, lors de l'appel de `changeDouble`, la valeur de `x`, à savoir `0.4` est recopiée dans la variable `a` (le paramètre formel). Ensuite, cette variable est modifiée, mais ceci n'a aucune incidence sur la variable `x` de la méthode `main`. Le problème est maintenant de savoir ce qui se passe quand le paramètre d'une méthode est un objet.

Comme nous l'avons déjà indiqué à la section 7.3.5, **on ne transmet jamais un *objet* d'une méthode à une autre : seules les références sont transmises**. Quand une méthode comme `changeString` possède un paramètre formel de type `String` (c'est-à-dire que le paramètre formel va faire référence à un objet), il y a, comme pour les types fondamentaux, création d'une variable classique (dans la pile). Cette variable est initialisée avec la *référence* qui forme le paramètre effectif lors de l'appel. Donc, le paramètre formel fait référence **au même objet** que le paramètre effectif. La figure 7.14 représente l'évolution de la mémoire lors de l'exécution de la méthode `changeString`. Le premier dessin représente la mémoire juste au début de la méthode, quand la référence a été recopiée. On voit bien que les variables `a` et `s` font référence au même objet. Le second dessin montre le résultat de `a=a+" et tata";` qui crée un nouvel objet auquel `a` va faire référence, sans aucune incidence sur `s`. Comme ce mode de transfert est le même pour tous les types objet, il est

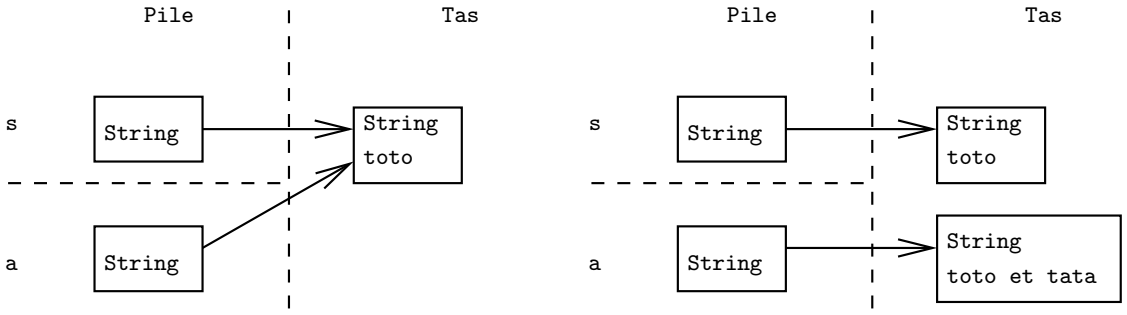

FIG. 7.14 – Passage d'un paramètre `String`

évident que le comportement va être différent si les objets sont modifiables. En effet, comme le paramètre formel fait référence *au même objet* que la variable d'origine de la méthode appelante, toute modification de l'objet par la méthode appelée a une répercussion dans la méthode appelante. C'est exactement ce qui se passe dans le programme présenté ici. Les variables `sb` et `a` (de la méthode `changeStringBuffer`) font référence au même `StringBuffer`. La modification de cet objet par `a.append(" et tata");` sera donc visible dans la méthode `main`. Ce comportement est illustré par la figure 7.15.

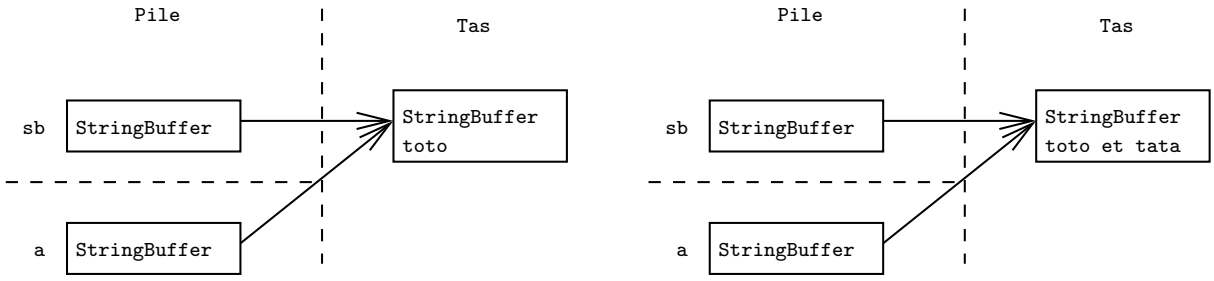

FIG. 7.15 – Passage d'un paramètre `StringBuffer`

REMARQUE

Nous remarquons que les figures 7.14 et 7.15 que la barrière qui sépare les zones mémoires des méthodes ne se prolonge pas dans le tas. En effet, comme nous l'avons indiqué à la section 7.3.3,

le tas ne peut pas être découpé en zones. Il est constitué d'un seul bloc et il n'est pas possible de protéger une partie des opérations réalisées par une méthode : dès qu'on possède une référence vers un objet, on peut manipuler cet objet depuis n'importe quelle méthode.

7.4.8 Retour sur les comparaisons

Cas général

Il faut bien comprendre que la manipulation par référence des objets a des conséquences sur **toutes** les opérations faisant intervenir les objets. C'est le cas en particulier des **comparaisons**, que nous avons déjà étudiées à la section 7.3.7 avec l'exemple des `String`s. Il est bien sûr impossible en général de définir un *ordre* sur des objets. Par contre, on peut plus facilement dire si un objet est *égal* à un autre. Plus précisément, si `a` et `b` sont deux variables d'un type objet, les expressions `a==b` et `a !=b` sont parfaitement correctes. Le point délicat est que la comparaison se fait **au niveau des références**. Ceci signifie qu'on ne compare pas du tout les valeurs représentées par les objets, mais au contraire les références qui permettent d'accéder aux objets. Il faut bien comprendre qu'**à chaque objet correspond une référence distincte**, même si deux objets représentent la même valeur apparente (par exemple la même chaîne de caractères). De ce fait, le phénomène des **jumeaux** évoqué à la section 7.3.7 s'applique aux objets en général et la comparaison des références permet de distinguer deux jumeaux.

Voici un exemple de ce phénomène appliqué aux `StringBuffer`s :

Exemple 7.37 :

On considère le programme suivant :

ComparaisonStringBuffer

```
1 public class ComparaisonStringBuffer {
2   public static void main(String[] args) {
3     StringBuffer a=new StringBuffer("Toto");
4     StringBuffer b=new StringBuffer("Toto");
5     System.out.println(a==b);
6     System.out.println(a.equals(b));
7   }
8 }
```

L'affichage produit est le suivant :

Affichage

```
false
false
```

Le premier affichage n'est pas étonnant : chaque utilisation du constructeur (lignes 3 et 4) produit un objet différent. Les deux `StringBuffer`s obtenus représentent donc la même chaîne, mais ils sont distincts.

Le deuxième affichage est plus étonnant, car il peut sembler en contradiction avec la section 7.3.7 : nous avons vu dans cette section que la méthode d'instance `equals` des `String`s compare les chaînes de caractères représentées par les objets intervenants. En fait, comme nous l'avions indiqué au moment de la présentation de cette méthode, seules certaines classes permettent une comparaison des valeurs. Pour les `StringBuffer`s, ce n'est pas le cas : la méthode `equals` est ici strictement équivalente à l'opérateur `==`. Nous reviendrons sur ce point au chapitre 9.

REMARQUE

Insistons sur le fait que, comme toutes les autres conséquences de la manipulation par référence, l'interprétation de la comparaison s'applique à **tous** les types objet.

Cas particulier des `String`

Pour les `String`s, l'interprétation est malheureusement un peu plus complexe. Étudions en effet le programme suivant :

ComparaisonString

```
1  public class ComparaisonString {
2    public static void main(String[] args) {
3      String a="Toto";
4      String b="Toto";
5      String c="To"+"to";
6      System.out.println(a==b);
7      System.out.println(a==c);
8      System.out.println(b==c);
9      String d="To";
10     String e="to";
11     c=d+e;
12     System.out.println(a==c);
13     System.out.println(b==c);
14   }
15 }
```

L'affichage obtenu est :

AFFICHAGE

```
true
true
true
false
false
```

Cet affichage est en contradiction avec la règle de comparaison des références. En effet, les objets désignés par `a`, `b` et `c` sont *a priori* distincts et on ne devrait pas avoir d'affichage `true` (dans cette optique, les deux derniers affichages, correspondant aux lignes 12 et 13 du programme, sont cohérents).

En fait, `Java` réalise une simplification des occurrences des valeurs littérales de type `String` (c'est-à-dire des textes entre guillemets). Dans un programme, le compilateur analyse tous les textes et ne conserve qu'une occurrence de chacun d'eux. Dans le programme précédent, cela signifie que les deux occurrences de `"Toto"` (lignes 3 et 4) feront références **à un seul objet** `String` correspondant au texte `Toto`. De plus, le compilateur inclut dans son analyse les **expressions constantes** (cf section 2.3.5). Toute expression produisant une chaîne de caractères et ne faisant pas intervenir de variable est évaluée à la compilation et est traitée comme les valeurs littérales lors de la suppression des occurrences multiples. De ce fait, l'expression `"To"+"to"` de la ligne 5 est évaluée en `"Toto"` et `c` fera donc référence à l'unique objet `String` correspondant au texte `Toto`. Ainsi les variables `a`, `b` et `c` contiennent-elles la même référence vers cet objet. Ceci explique les trois `true` affichés par le programme.

A la ligne 11, `c` fait référence à un objet `String` obtenu en mettant bout à bout le contenu des objets auxquels `d` et `e` font référence. L'expression de la ligne 11 n'étant pas une expression constante, le calcul a lieu lors de l'exécution du programme et il y a bien production d'un nouvel objet `String` correspondant au texte `Toto`. De ce fait, les deux `false` qui s'affichent à cause des lignes 12 et 13 sont parfaitement logiques : `c` fait référence à un nouvel objet, différent de celui auquel `a` et `b` font référence.

REMARQUE

Le mécanisme mis en œuvre par le compilateur `Java` est très complexe et il n'est pas nécessaire de le retenir parfaitement, d'autant plus que le système utilisé ne s'applique qu'aux objets de type `String` et que la description que nous en donnons est une simplification importante de la réalité.

7.5 Conseils d'apprentissage

Le présent chapitre n'est pas organisé de la même façon que les chapitres précédents. Plutôt que de présenter des outils, puis d'expliquer leurs applications possibles, nous avons préféré partir de problèmes concrets (la manipulation de texte, puis l'efficacité de celle-ci) pour introduire les concepts délicats liés à la manipulation des objets. Il en résulte un chapitre dans lequel les informations concernant tous les types objet sont mélangés avec des éléments qui s'appliquent avant tout aux types `String` et `StringBuffer`. Or ces deux types sont intéressants à la fois en tant que types objet centraux en `Java`, mais aussi comme première application réaliste des techniques algorithmiques (boucles, sélection, etc.) étudiées dans les chapitres précédents. Le lecteur peut donc éprouver des difficultés à bien isoler la partie concernant la manipulation des objets et les nouvelles perspectives algorithmiques ouvertes par les opérations possibles sur les `String`s et `StringBuffer`s. Pour faciliter l'apprentissage des notions générales aux objets, voici quelques conseils :

- Il est impératif de bien comprendre que les objets sont manipulés par **référence**, c'est la véritable nouveauté :
  - quand on manipule une variable d'un type objet quelconque, celle-ci contient toujours une référence (éventuellement `null`) qui désigne l'objet qu'elle permet d'utiliser ;
  - le point le plus important à retenir est que **la sémantique des variables reste totalement inchangée** : une affectation se traduit par la copie du contenu de la variable de départ (c'est-à-dire de la référence), une comparaison par `==` se traduit par une comparaison des contenus (c'est-à-dire des références), un passage de paramètre correspond à une copie de la référence, une définition du résultat d'une méthode s'interprète aussi par une copie de la référence ;
  - la principale difficulté vient du fait que la manipulation des objets est **indirecte** alors qu'elle semble **directe** : ceci se traduit par le phénomène complexe des **effets de bord**.
- On doit aussi retenir que la création des objets est dynamiques : on doit impérativement passer par un **constructeur** et l'instruction `new` pour obtenir une référence sur un nouvel objet (sauf dans le cas exceptionnel des `String`s).
- Enfin, excepté la concaténation des `String`s, il faut noter que les opérations sur les objets passent toujours par les **méthodes d'instance** dont la sémantique est très proche de celle des méthodes de classe, mais dont le protocole d'appel est différent : on doit impérativement fournir un **objet appelant**.

Pour bien acquérir les éléments présentés dans ce chapitre, il est utile de commencer par une première étude se focalisant sur l'application proposée des objets, les types `String` et `StringBuffer`. On peut ensuite relire le chapitre en cherchant à généraliser les notions pour bien comprendre qu'elles s'appliquent à tout type objet.

Notons qu'après ce chapitre, le lecteur est théoriquement capable d'utiliser n'importe quel type objet, en se basant bien entendu sur sa documentation (voir [12]). C'est la raison pour laquelle nous ne donnerons pas dans cet ouvrage une documentation plus complète des diverses classes utiles en `Java` (comme bien entendu les classes `String` et `StringBuffer`). Nous ne pouvons que conseiller a l'apprenti programmeur la lecture de la documentation de `Java`, en commençant par les classes déjà abordées dans ce chapitre. Ces classes sont très riches et la manipulation de leurs méthodes d'instance permettra au lecteur de travailler les techniques algorithmes importantes. Dans un deuxième temps, le lecteur pourra s'intéresser à des classes importantes en `Java`, comme par exemple `Random` qui permet la gestion des nombres aléatoires, `Locale` qui permet d'adapter un programme aux particularités de la langue de l'utilisateur (représentation des nombres, des dates, etc.), `BigInteger` et `BigDecimal` qui permettent le calcul en précision arbitraire (autant de chiffres après la virgule que nécessaire), etc.

CHAPITRE 8

Les tableaux

Sommaire

Introduction et motivations

La principale utilisation des tableaux est la représentation informatique des *listes*. Il est en effet naturel par exemple de vouloir manipuler les notes d'un étudiant pour calculer sa moyenne. Or, il nous faut déclarer une variable pour chaque note. Donc, si l'étudiant a 20 notes, il faut déclarer 20 variables différentes, faire 20 saisies, etc. ce qui est très fastidieux.

Le problème vient essentiellement du fait qu'il n'est pas possible d'écrire une boucle qui manipulerait successivement plusieurs variables différentes, plus précisément qui changerait de variable étudiée à chaque itération de la boucle. On ne peut donc pas écrire facilement un programme qui calcule $\frac{1}{n}\sum_{k=1}^{n} u_k$ si chaque u_k est stocké dans une variable différente.

De la même façon, il n'est pas possible de créer une quantité de variables qui dépend d'une information fournie par l'utilisateur. Donc, si un élève suit par exemple une matière optionnelle, il faut soit faire saisir la note pour tout le monde, soit ne pas en tenir compte.

Toutes ces limitations amènent à étudier de nouveaux objets appelés des **tableaux**. Un tableau est en quelque sorte une liste de variables (ou une suite finie). Grâce un tableau, on peut manipuler aisément plusieurs variables au sein d'une boucle ce qui va résoudre les problèmes évoqués dans cette introduction.

Nous commencerons ce chapitre par étudier comment déclarer et manipuler des tableaux. En tant qu'objets, les tableaux posent quelques problèmes que nous étudierons. Nous verrons ainsi comment représenter l'occupation de la mémoire par un tableau, comment manipuler globalement un tableau (en particulier par l'intermédiaire de **valeurs littérales** et des méthodes de copie et de **clonage**).

Dans un deuxième temps, nous présenterons des utilisations avancées des tableaux, en commençant par les **tableaux multidimensionnels** qui permettent entre autre de représenter des objets mathématiques comme les matrices. Nous terminerons ce chapitre par l'étude des **tableaux d'objets**.

8.1 Premières manipulations

8.1.1 Déclaration

Nous souhaitons donc pouvoir manipuler une liste de valeurs, c'est-à-dire un n-uplet, de la forme $(x_0, x_1, \ldots, x_{n-1})$. Comme pour les chaînes de caractères, il semble logique de désigner sous un même type des n-uplets de même nature mais pas obligatoirement de même longueur. Si on considère par exemple un relevé de températures, la technique pour calculer la température maximale est la même quel que soit le nombre de températures que comporte le relevé. Or, on ne pourra utiliser le même programme pour travailler sur 10, 20 ou un nombre arbitraire de températures que si deux relevés de taille différentes sont représentés par le même type.

Mais dans ce cas, une liste de valeurs n'est plus une valeur classique : c'est donc un *objet*, au même titre que les chaînes de caractères. Comme tout objet, une liste de valeurs va donc être créée dans le tas et manipulée grâce à une variable qui lui fera référence depuis la pile[1].

Pour déclarer une telle variable, on utilise une construction syntaxique nouvelle illustrée par l'exemple suivant :

```
double[] températures;
```

La seule différence avec la déclaration d'une variable de type `double` est la présence d'une paire de crochets après `double`. Cette déclaration signifie que la variable `températures` fait référence à un objet de type **tableau de** `doubles`. Notons que la dénomination **tableau** est spécifique à l'informatique. Intuitivement, il s'agit plutôt d'une **liste** de `double`s, ou pour adopter un vocabulaire plus mathématique, une **suite finie**. Nous remarquons que, comme prévu, le nombre d'éléments dans cette liste n'apparaît pas dans le type.

REMARQUE

Il est possible de remplacer la déclaration `double[] températures ;` par :

```
double températures[];
```

Les deux déclarations ont exactement le même sens.

8.1.2 Création

Après avoir déclaré une variable faisant référence sur une liste, il faut créer cette liste, de la même façon qu'on doit créer un `StringBuffer` avant de l'utiliser. Pour cela, nous utilisons l'instruction `new` de la façon suivante :

```
températures=new double[10];
```

Dans cette instruction, le nombre 10 donne la taille de la liste qui va être créée. Le processeur crée ici une liste de 10 éléments à laquelle la variable `températures` fait référence.

[1] Pour bien comprendre les tableaux, il faut donc avoir complètement assimilé le chapitre précédent.

REMARQUE

La création d'un tableau est très similaire à celle des autres objets. En effet, le `new` est suivi par le type du tableau avec, entre les crochets, un paramètre qui indique la taille et qui s'apparente donc aux paramètres des constructeurs évoqués à la section 7.4.3.

8.1.3 Manipulation

Pour manipuler une chaîne de caractères représentée par un `StringBuffer`, nous avons utilisé des méthodes d'instance permettant respectivement de lire (`charAt`) un caractère et de le modifier (`setCharAt`). Pour les *tableaux*, on utilise une technique simplifiée basée sur la notation utilisant les crochets. Nous pouvons en effet représenter une liste à n éléments comme un n-uplet, $(x_0, x_1, \ldots, x_{n-1})$. Par analogie, les 10 éléments créés par le `new` qui vient d'être présenté sont appelés `températures[0]`, `températures[1]`, etc. jusqu'à `températures[9]`.

Chaque "variable" ainsi désignée fait directement référence à l'emplacement de la mémoire utilisée par l'objet (dans le tas) pour représenter l'élément correspondant. En d'autres termes, pour dire que le premier élément de la liste de température vaut -5, il suffit d'écrire :

```
températures[0]=-5;
```

De même, pour afficher la 5-ième température de la liste, il suffit de faire :

```
System.out.println(températures[4]);
```

Le principal avantage d'une variable *de type tableau* par rapport à un ensemble de variables classiques est que le nom de chaque élément de la liste qu'elle représente peut être *calculé* : en effet, le numéro de l'élément est tout simplement une valeur entière. Il est donc possible d'écrire par exemple `températures[1+2]=-5 ;`, ou plus généralement si `i` est une variable de type `int`, `températures[i]=-5 ;`, à condition bien sûr que la valeur de `i` soit comprise au sens large entre 0 et 9.

REMARQUE

Comme pour les `String`s (voir la section 7.2.3), une tentative d'accès en dehors des indices acceptables pour un tableau (par l'exemple l'élément 5 dans un tableau de 4 éléments) provoque l'arrêt du programme, qui affiche un message complexe comprenant le texte `ArrayIndexOutOfBoundsException`.

Exemple 8.1 :

Voici un exemple de programme qui tente d'accéder à un élément qui n'existe pas dans un tableau :

OutOfRangeTab

```
1 public class OutOfRangeTab {
2   public static void main(String[] args) {
3     double[] tab=new double[10];
4     tab[10]=2.5;
5   }
6 }
```

Le programme compile sans aucun problème. Par contre, à l'exécution, on obtient l'affichage suivant :

Erreur d'exécution

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at OutOfRangeTab.main(OutOfRangeTab.java:4)
```

8.1.4 Un exemple plus complet

Considérons le programme suivant :

TableauElementaire

```
1  import dauphine.util.*;
2  public class TableauElementaire {
3    public static void main(String[] args) {
4      Console.start();
5      double[] notes;
6      notes=new double[5];
7      for(int i=0;i<5;i++) {
8        System.out.print("Note "+i+" = ");
9        notes[i]=Console.readDouble();
10     }
11     for(int i=0;i<5;i++) {
12       System.out.println("Note "+i+" = "+notes[i]);
13     }
14   }
15 }
```

Il illustre parfaitement l'utilisation d'un tableau au sein d'une boucle : la variable `notes` est une liste de 5 notes. Ces notes sont demandées à l'utilisateur grâce à un simple boucle qui utilise le fait que les éléments sont accessibles au moyen de leur numéro (ici `notes[i]`). Une boucle similaire affiche ensuite les valeurs saisies par l'utilisateur.

Voici un affichage possible quand on exécute le programme :

Affichage

```
Note 0 = 10
Note 1 = 15
Note 2 = 12
Note 3 = 17
Note 4 = 16
Note 0 = 10.0
Note 1 = 15.0
Note 2 = 12.0
Note 3 = 17.0
Note 4 = 16.0
```

8.2 Cas général de tableau et vocabulaire associé

8.2.1 Type tableau

Soit *T* un type quelconque (représentant un type fondamental ou un type objet). Pour représenter une suite finie (une liste) d'éléments de type *T*, on définit un type **tableau de** T (aussi appelé **liste de** T). Ce type est un **type objet**. Pour manipuler un objet de type tableau de *T*, on utilise une variable qui contient une *référence* sur cet objet. La variable est déclarée par l'instruction suivante :

T[] *identificateur*;

ou par :

T *identificateur*[];

8.2.2 Création et manipulations élémentaires

La création d'un objet de type tableau de *T* correspondant à `n` éléments se fait par l'instruction suivante (le résultat devant être placé dans une variable appropriée, appelée *identificateur*) :

identificateur=new *T*[n];

On peut alors accéder aux éléments de l'objet en utilisant la *variable* *identificateur*[i], pour `i` une valeur entière comprise au sens large entre `0` et `n-1`. Il est important de noter que les variables *identificateur*[i] doivent être considérées comme *contenues* dans l'objet tableau. On les appelle les **cases** du tableau.

8.2.3 Longueur

On peut obtenir le nombre d'éléments (ou le nombre de cases) de l'objet auquel une variable *identificateur* fait référence en écrivant *identificateur*.`length`, qui est une valeur entière (de type `int`). Ce nombre d'éléments s'appelle aussi la **longueur** ou la **taille** du tableau.

REMARQUE

Dans l'écriture précédente, `length` n'est pas suivie d'une paire de parenthèses, contrairement au cas (par exemple) des `String`, car, pour des raisons techniques complexes (essentiellement des problèmes d'efficacité), ce n'est pas une méthode, mais une constante d'instance. Pour simplifier, on peut dire qu'une constante d'instance (aussi appelée une constante d'objet), joue pour un objet le même rôle que celui joué par une constante de classe pour une classe.

Il très important de noter qu'il n'existe **aucun moyen** de modifier la longueur d'un objet tableau après sa création. On pourrait croire qu'il suffit d'affecter une nouvelle valeur à `length`, mais le compilateur refuse toute tentative de ce genre, comme l'illustre l'exemple suivant :

Exemple 8.2 :

On tente la manipulation suivante :

LongueurFixe

```
1 public class LongueurFixe {
2   public static void main(String[] args) {
3     int[] t=new int[5];
4     // tentative de réduction
5     t.length=4;
6   }
7 }
```

Le compilateur refuse le programme et affiche le message suivant :

Erreur de compilation

```
LongueurFixe.java:5: cannot assign a value to final variable length
    t.length=4;
     ^
1 error
```

8.3 Exemples d'application

8.3.1 Calcul d'une moyenne

Considérons le programme suivant :

Moyenne

```
1  import dauphine.util.*;
2  public class Moyenne {
3    public static void main(String[] args) {
4      Console.start();
5      int nb;
6      do {
7        System.out.print("Nombre de valeurs : ");
8        nb=Console.readInt();
9      } while (nb<=0);
10     double[] températures=new double[nb];
11     for(int i=0;i<températures.length;i++) {
12       System.out.print("Température "+i+" : ");
13       températures[i]=Console.readDouble();
14     }
15     double moyenne=0;
16     for(int i=0;i<températures.length;i++)
17       moyenne=moyenne+températures[i];
18     moyenne=moyenne/températures.length;
19     System.out.println("Moyenne = "+moyenne);
20   }
21 }
```

Ce programme utilise complètement les possibilités des tableaux et en particulier le fait que le tableau peut être créé dynamiquement : on peut donc choisir la taille de celui-ci au moment de *l'exécution* du programme (en réaction aux demandes de l'utilisateur). Ceci nous permet ici de créer un tableau pour stocker un certain nombre de températures, inconnu avant l'exécution du programme. Ensuite, en utilisant la longueur du tableau, on peut faire saisir les valeurs qui le constituent avec une boucle. De la même façon, on peut ensuite calculer la moyenne de ces valeurs : pour ce faire, on définit la moyenne par $m = \frac{1}{n}\sum_{i=0}^{n-1} t_i$, où les t_i forment la liste finie décrite par le tableau. On remarque que mn est en fait le terme u_n de la suite définie par $u_i = u_{i-1} + t_i$ (et $u_{-1} = 0$). On reconnaît la mise sous forme récurrente d'un calcul et on sait donc calculer mn par une simple boucle `for`. Il ne reste plus qu'à diviser le résultat par n pour obtenir la moyenne des températures. Voici un exemple d'affichage possible pour ce programme :

Affichage

```
Nombre de valeurs : 3
Température 0 : 23
Température 1 : 45
Température 2 : 34
Moyenne = 34.0
```

Le programme obtenu est correct, mais il n'est pas très satisfaisant car tout est situé dans la méthode principale. Voici une version mieux organisée :

Moyenne2

```
1  import dauphine.util.*;
2  public class Moyenne2 {
3    public static double[] readTab () {
4      int nb;
5      do {
6        System.out.print("Nombre de valeurs : ");
7        nb=Console.readInt();
8      } while (nb<=0);
9      double[] liste=new double[nb];
10     for(int i=0;i<liste.length;i++) {
11       System.out.print("Température "+i+" : ");
12       liste[i]=Console.readDouble();
13     }
14     return liste;
15   }
16   public static double computeMean(double[] liste) {
17     double moyenne=0;
18     for(int i=0;i<liste.length;i++)
19       moyenne=moyenne+liste[i];
20     return moyenne/liste.length;
21   }
22   public static void main(String[] args) {
23     Console.start();
24     double[] températures=readTab();
25     System.out.println("Moyenne = "+computeMean(températures));
26   }
27 }
```

On remarque qu'il est donc possible de *renvoyer* un tableau avec une méthode ainsi que d'avoir des paramètres formels de type tableau. La réécriture du programme avec trois méthodes montre l'importance de l'utilisation de `length` : cette valeur est liée au tableau et quand on utilise un tableau comme résultat ou comme paramètre d'une méthode, *la longueur est transmise avec le tableau*. Nous avions déjà rencontré ce comportement avec les chaînes de caractères, mais nous n'avions pas évoqué son importance à ce moment. Il est ici rigoureusement impossible d'écrire simplement une méthode qui renvoie un tableau et sa taille *séparément et simultanément* (car une telle méthode aurait deux résultats, ce qui est impossible). De ce fait, si on ne disposait pas de la taille du tableau avec celui-ci, on ne pourrait pas écrire de méthode renvoyant un tableau.

Voici un exemple d'interaction possible avec le programme :

Affichage

```
Nombre de valeurs : 6
Température 0 : 20
Température 1 : 19
Température 2 : 21
Température 3 : 22
Température 4 : 18
Température 5 : 20
Moyenne = 20.0
```

REMARQUE

Le programme `Moyenne2` comporte une méthode de saisie (la méthode `readTab`). Cette méthode correspond exactement à la règle proposée à la section 6.5.4 : les instructions de saisie apparaissent exclusivement dans les méthodes dont l'unique but est de réaliser une saisie.

8.3.2 Moyenne pondérée

Nous pouvons poursuivre l'exemple qui vient d'être étudié en proposant un programme qui calcule une moyenne pondérée : on va donc demander le nombre de notes, les coefficients et enfin les notes. Puis on calculera la moyenne. Voici la solution proposée :

Moyenne3

```
1  import dauphine.util.*;
2  public class Moyenne3 {
3    public static double[] readCoeff() {
4      int nb;
5      do {
6        System.out.print("Nombre de coefficients : ");
7        nb=Console.readInt();
8      } while (nb<=0);
9      double[] liste=new double[nb];
10     for(int i=0;i<liste.length;i++) {
11       System.out.print("Coefficient "+(i+1)+" : ");
12       liste[i]=Console.readDouble();
13     }
14     return liste;
15   }
16   public static double[] readNotes(int nb) {
17     double[] liste=new double[nb];
18     for(int i=0;i<liste.length;i++) {
19       System.out.print("Note "+(i+1)+" : ");
20       liste[i]=Console.readDouble();
21     }
22     return liste;
23   }
24   public static double computeWeightedMean(double[] coeff,double[] notes) {
25     double moyenne=0;
26     double totalCoeff=0;
27     for(int i=0;i<coeff.length;i++) {
28       moyenne=moyenne+coeff[i]*notes[i];
```

```
29        totalCoeff=totalCoeff+coeff[i];
30      }
31      return moyenne/totalCoeff;
32    }
33    public static void main(String[] args) {
34      Console.start();
35      double[] coefficients=readCoeff();
36      double[] notes=readNotes(coefficients.length);
37      System.out.println("Moyenne = "+computeWeightedMean(coefficients,notes));
38    }
39  }
```

La méthode `readCoeff` est quasiment identiquement à la méthode `readTab` du programme précédent et consiste simplement à saisir la liste des coefficients. La méthode `readNotes` prend elle un paramètre entier : le nombre de notes à saisir. En effet, si on utilisait deux fois la méthode `readCoeff`, on pourrait obtenir un nombre différent de valeurs numériques, ce qui rendrait impossible le calcul de la moyenne pondérée. La méthode `computeWeightedMean` est très proche de `computeMean`. Le principe est simplement de calculer la moyenne pondérée des notes n_i pour les pondérations c_i définie par :

$$m = \frac{\sum_{i=0}^{n-1} c_i n_i}{\sum_{i=0}^{n-1} c_i}$$

Une fois de plus, on utilise la mise sous forme récurrente de $m = \frac{u_n}{v_n}$. La seule "astuce" est de calculer u_k et v_k en parallèle (c'est à dire en même temps dans une seule boucle).

Voici un exemple d'interaction avec le programme obtenu :

AFFICHAGE

```
Nombre de coefficients : 3
Coefficient 1 : 2
Coefficient 2 : 3
Coefficient 3 : 1
Note 1 : 15
Note 2 : 12
Note 3 : 17
Moyenne = 13.833333333333334
```

8.3.3 Algorithmes et tableaux

Dans les exemples précédents, nous avons écrit directement les programmes malgré leur (très) relative complexité algorithmique. En fait, il nous manquait une technique simple d'écriture d'algorithmes manipulant les tableaux. Si on revient à l'idée principale des algorithmes, à savoir décrire comment résoudre une tâche sans faire référence à un langage de programmation, il semble clair qu'on ne va pas parler de tableau mais plutôt de liste, c'est-à-dire qu'on va manipuler l'objet mathématique plutôt que l'objet informatique.

Prenons comme exemple le calcul de la moyenne des éléments contenus dans un tableau. En fait, il s'agit de calculer la moyenne des réels de la liste $(u_0, u_1, \ldots, u_{n-1})$, où n désigne le nombre d'éléments dans la liste. Nous écrirons donc l'algorithme ainsi :

Données :

– une liste de n réels, $l = (u_0, u_1, \ldots, u_{n-1})$.

Résultat : la moyenne des éléments de la liste l.

1. initialiser `somme` à 0
2. pour `i` allant de 0 à $n-1$
   Ajouter $u_{\texttt{i}}$ à `somme`
3. Résultat : `somme`$/n$

Encore une fois, nous obtenons un algorithme assez différent de la méthode qui calcule la moyenne.

REMARQUE

De façon générale, il est important de bien comprendre que la différence entre les algorithmes et les méthodes doit être cultivée. En effet, être capable d'écrire un algorithme utilisant des listes prouve qu'on est capable d'avoir une réflexion abstraite sur la programmation, qu'on a vraiment compris les possibilités de manipulation des listes induites par la structure de tableau.

De plus, les livres et articles qui traitent d'algorithmique présentent toujours les programmes de façon abstraite (sous forme d'algorithme), en général pour éviter de donner une version de la solution proposée trop spécifique à un langage de programmation en particulier. Il faut donc être capable de traduire en `Java` (par exemple) des algorithmes abstraits.

8.3.4 Application dans la méthode `main`

Nous avons vu à la section 6.1.3 qu'une application `Java` correspond au minimum à une classe comportant une méthode `main`. Cette méthode doit obligatoirement avoir un unique paramètre (dont le nom est quelconque) dont le type est `String[]`. Ce paramètre correspond donc à un tableau de chaînes de caractères. Ce tableau correspond à *l'environnement* du programme.

En fait, quand on lance un programme depuis un interpréteur de commandes (un *shell* sous Unix et une fenêtre MsDos sous Windows 9x), on peut lui donner des *paramètres*. Si on souhaite par exemple renommer un fichier (sous Unix), on écrit `mv toto toto-new`. Le programme appelé est `mv`, avec les paramètres `toto` et `toto-new`. De la même façon, on peut transmettre des paramètres à un programme `Java` appelé depuis un interpréteur de commande. Ces paramètres sont copiés avant le lancement du programme dans un tableau de `String`, chaque paramètre correspondant à une chaîne de caractères. Le tableau des paramètres est transmis comme paramètre effectif à la méthode `main`, ce qui permet à celle-ci d'avoir un comportement qui dépend justement de la valeur de ces paramètres.

Dans le cadre de ce cours, nous n'utiliserons pas cette possibilité, mais il nous a semblé important de lever le voile sur ce paramètre formel particulier de la méthode `main`, d'autant plus que dans la pratique, en particulier sous Unix, il est très fréquent de transmettre à un programme des paramètres quand on le lance.

8.4 Représentation en mémoire des tableaux

Un tableau est manipulé par référence, comme le sont tous les objets. Il faut bien comprendre que c'est la variable qui permet de manipuler l'objet tableau qui est une référence, ce qui a des conséquences complexes sur le comportement des tableaux.

8.4.1 Représentation de la mémoire

Comme nous l'avons vu à partir de la section 7.3.4, la représentation graphique de la mémoire permet de mieux comprendre la manipulation par référence. Comme les tableaux apportent leur

lot de problèmes, nous allons introduire une représentation spécifique de ceux-ci avant de passer à des exemples concrets.

Pour ce faire, considérons la création de tableau suivante :

```
int[] x=new int[3];
x[0]=2;
x[1]=5;
x[2]=10;
```

Représentons maintenant l'état de la mémoire après l'exécution de la quatrième et dernière affectation. La figure 8.1 donne une représentation possible de la mémoire. C'est une représentation très

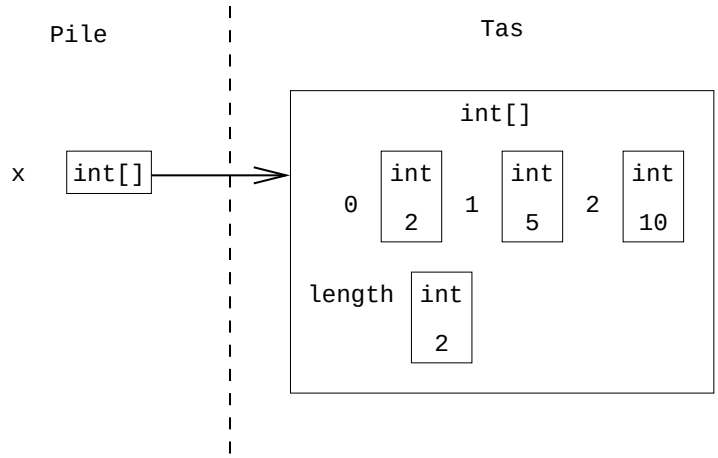

FIG. 8.1 – Représentation d'un tableau en mémoire : version complète

complète :

- la constante `length` est indiquée (ainsi que son type) ;
- on insiste sur le fait que chaque case du tableau est une variable de type `int`, contenue dans le tableau ;
- on précise le nom de chaque variable, c'est-à-dire son indice compris entre 0 et la longueur du tableau (non incluse).

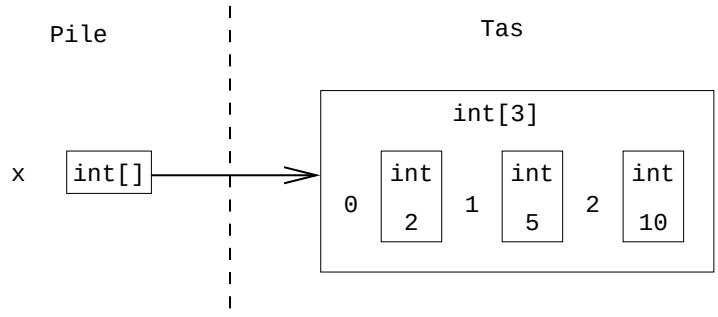

FIG. 8.2 – Représentation d'un tableau en mémoire : version simplifiée

On peut simplifier la représentation, en particulier au niveau de l'information sur la longueur du tableau, comme l'illustre la figure 8.2 : on a supprimé la constante `length` et on a indiqué dans le type du tableau la taille de celui-ci, en référence à la construction utilisée pour créer l'objet. On

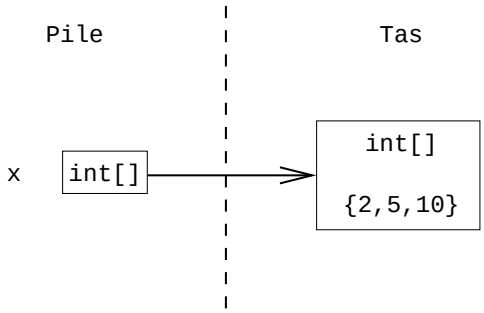

FIG. 8.3 – Représentation d'un tableau en mémoire : version minimale

peut aussi se passer complètement de la mention de la longueur, celle-ci étant implicite (car on représente toutes les cases du tableau).

On peut enfin adopter une version minimaliste de la représentation en mémoire, comme l'illustre la figure 8.3. Cette représentation se base sur la notion de valeur littérale de type tableau, que nous présenterons à la section 8.5.2 (pour simplifier, disons qu'on donne la liste qui constitue le contenu du tableau en séparant les éléments par des virgules et en encadrant la liste par des accolades). Cette représentation minimale est particulièrement adaptée à la manipulation de tableau dont les éléments sont d'un type fondamental. Quand on travaille avec des tableaux d'objets, les manipulations sont beaucoup plus complexes et il est très difficile de bien comprendre ce qui se passe sans utiliser une représentation plus complète, comme nous le verrons dans les sections 8.6 et 8.7.

8.4.2 Effet de bord

Nous avons déjà vu la notion d'effet de bord pour les `StringBuffer`. Le principe est d'avoir deux variables qui font référence au même objet. On peut alors utiliser la première pour modifier l'objet et observer l'effet de cette modification grâce à la seconde : on a donc l'impression d'avoir modifié la seconde variable comme par magie, sans utiliser son nom. Voici un exemple élémentaire avec les tableaux :

Exemple 8.3 :

BordTableau

```
1  public class BordTableau {
2    public static void main(String[] args) {
3      int[] a,b;
4      a=new int[3];
5      a[0]=0;
6      a[1]=1;
7      a[2]=2;
8      b=a;
9      System.out.println(b[0]+", "+b[1]+", "+b[2]);
10     b[0]=-1;
11     System.out.println(a[0]+", "+a[1]+", "+a[2]);
12   }
13 }
```

Ce programme produit l'affichage suivant :

AFFICHAGE

```
0, 1, 2
-1, 1, 2
```

On voit donc que l'instruction `b=a` place dans `b` la même référence que dans `a` : les deux variables font donc référence au même objet, comme l'illustre la figure 8.4. Ensuite, comme

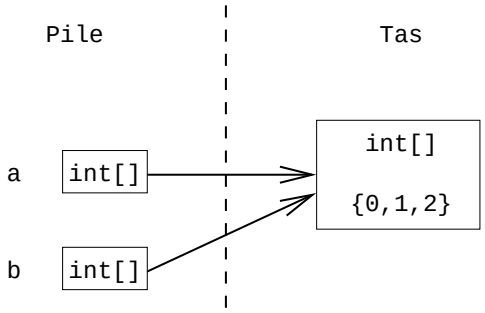

FIG. 8.4 – La mémoire juste après l'exécution de la ligne 8

prévu, la modification du tableau auquel `b` fait référence modifie aussi le tableau auquel `a` fait référence puisqu'il s'agit du même objet !

8.4.3 Utilisation dans les méthodes

Les remarques concernant les `StringBuffer` données dans la section 7.4.6 du chapitre 7 sont encore valables ici. Comme les tableaux sont manipulés par référence, une méthode peut modifier un objet auquel des variables d'une autre méthode font référence, provoquant de nouveau un effet de bord.

Étudions l'exemple suivant :

Exemple 8.4 :

On considère la classe suivante :

TableauMethode

```
1  public class TableauMethode {
2    public static void modifOne(int[] a) {
3      a[0]=1;
4    }
5    public static void modifTwo(int[] a) {
6      int[] b=new int[a.length];
7      for(int i=0;i<a.length;i++) {
8        b[i]=a[i];
9      }
10     b[0]=2;
11     a=b;
12   }
13   public static void display(int[] a) {
14     System.out.println("----------");
15     for(int i=0;i<a.length;i++) {
16       System.out.println("tab["+i+"]="+a[i]);
17     }
18     System.out.println("----------");
```

```
19    }
20    public static void main(String[] args) {
21      int[] test=new int[2];
22      test[0]=0;
23      test[1]=1;
24      display(test);
25      modifOne(test);
26      display(test);
27      modifTwo(test);
28      display(test);
29    }
30  }
```

Ce programme affiche les lignes suivantes :

AFFICHAGE

```
----------
tab[0]=0
tab[1]=1
----------
----------
tab[0]=1
tab[1]=1
----------
----------
tab[0]=1
tab[1]=1
----------
```

La méthode `display` se charge simplement de l'affichage des éléments du tableau qui lui est transmis en paramètre. La méthode `modifOne` essaye de modifier l'objet qui lui est transmis en paramètre. L'affichage indique que cette modification est un succès. Par contre, la tentative de modification effectuée par `modifTwo` est un échec.

Pour comprendre le problème, étudions plus précisément chaque méthode :

- `modifOne`
  Dans cette méthode très simple, on donne la valeur `1` au premier élément du tableau auquel la variable `a` (paramètre de la méthode) fait référence. Or, d'après les règles d'appel des méthodes, l'objet auquel `a` fait référence est le même tableau que celui auquel la variable `test` du programme principal fait référence. Donc, il y a bien ici effet de bord. La figure 8.5 montre l'effet de l'appel de la méthode `modifOne` sur la mémoire, et notamment l'utilisation du paramètre pour modifier l'objet référencé.
- `modifTwo`
  Dans cette méthode, on commence par créer un *nouveau* tableau manipulé par l'intermédiaire de la variable `b`. On recopie le contenu du tableau référencé par `a` dans `b` : on a donc à ce moment deux tableaux *distincts* dans la mémoire (dans le tas). On modifie alors le nouveau tableau puis on indique que `a` fait dorénavant référence à ce nouveau tableau. Mais ceci n'a aucune influence sur la variable `test` de la méthode `main` : en effet, les variables de différentes méthodes sont indépendantes et on n'a pas utilisé `a` pour modifier le tableau auquel `test` fait référence.

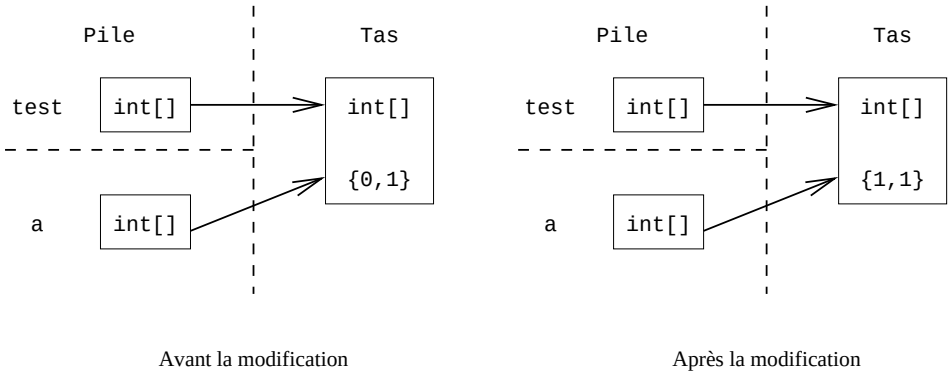

FIG. 8.5 – Effet sur la mémoire de l'appel de `modifOne`

La figure 8.6 montre l'effet de l'appel de la méthode `modifTwo` sur la mémoire. Elle illustre en particulier la création d'un nouvel objet.

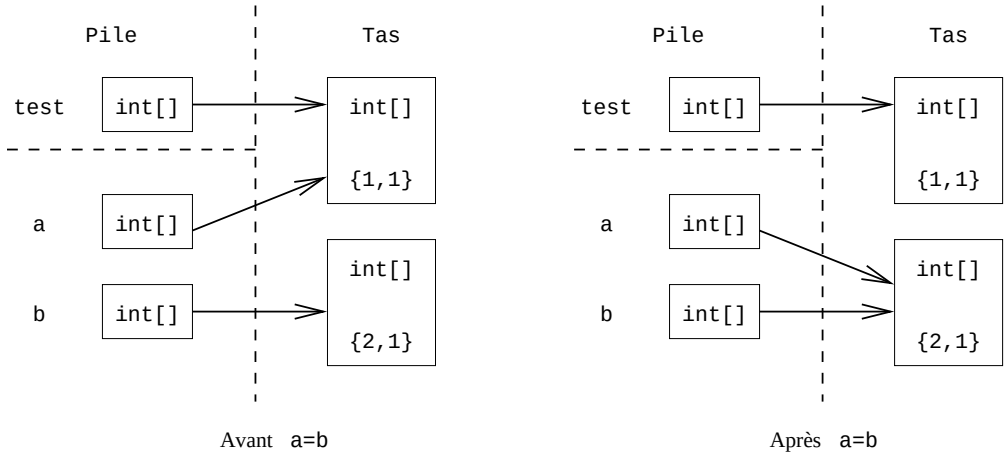

FIG. 8.6 – Effet sur la mémoire de l'appel de `modifTwo`

En résumé, on remarque qu'il est possible d'utiliser une méthode pour modifier les éléments d'un tableau qui lui est transmis en paramètre. Par contre, il est impossible de remplacer de tableau par un autre tableau.

REMARQUE

Le fait qu'on ne puisse pas remplacer un tableau par un autre a pour conséquence qu'on ne peut pas changer la **taille** d'un tableau par effet de bord. En effet, comme nous l'avons déjà dit, il n'existe aucune méthode d'instance des tableaux permettant de changer la taille du tableau appelant (contrairement par exemple aux méthodes `append` des `StringBuffer`s).

8.5 Manipulations globales des tableaux

8.5.1 Valeurs initiales dans un tableau

Nous savons depuis le chapitre 2 qu'une variable doit obligatoirement être initialisée avant de pouvoir être utilisée. Or dans la manipulation des tableaux, nous n'avons pas évoqué ce problème

car nous n'avons pas mentionné à ce moment les effets "secondaires" du `new`. En effet, quand on écrit la ligne suivante :

```
int [] tab=new int[10];
```

l'ordinateur ne se contente pas de créer un tableau possédant dix cases dans le tas (et de placer une référence vers ce tableau dans la variable `tab`). Il réalise aussi une initialisation des cases du tableau qui prennent toutes la valeur `0`, ce qui veut dire que les variables en question seront bien initialisées.

De façon générale, la création d'un tableau par un `new` provoque l'initialisation des cases de ce tableau à une valeur qui dépend du type des éléments du tableau. Voici une liste de ces valeurs initiales :

Type	valeur
`boolean`	`false`
`byte`	`0`
`char`	`'\u0000'`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0F`
`double`	`0.0`
objet	`null`

On remarque que pour tous les tableaux d'objets (par exemple un tableau de `String`), la valeur initiale de chaque case est la valeur spéciale `null` évoquée à la section 7.3.6. De ce fait, un tableau d'objets ne fait référence initialement à aucun objet (ce qui semble normal, car il faudrait que ces objets soient eux-mêmes créés par des appels à `new`). On voit donc ici une utilisation cruciale de la référence `null`, sans laquelle il serait très difficile de travailler avec les tableaux.

8.5.2 Valeurs littérales de type tableau

Nous avons vu dans les chapitres précédents qu'il était possible d'écrire des valeurs numériques (section 2.2.1) ou des chaînes de caractères (section 7.1.1) directement dans un programme. Ceci permet, par exemple, de donner une valeur initiale à une variable. Or, il est aussi possible de donner des valeurs littérales de type *tableau*, c'est-à-dire de donner une liste de valeurs à placer dans un tableau.

Pour ce faire, on écrit la liste des valeurs en les entourant d'une paire d'accolades et en les séparant par des virgules. Voici par exemple une liste de réels :

```
{0.1,0.2,-5.75,2.4}
```

Cette liste correspond à un tableau de type `double[]` de longueur 4. Les éléments sont bien entendu placés dans l'ordre dans le tableau correspondant. Voici un exemple simple de démonstration :

Exemple 8.5 :

Dans la méthode `main`, on utilise une valeur de type tableau de réels et une autre de type tableau de caractères.

TableauValeur

```
1  public class TableauValeur {
2    public static void main(String[] args) {
3      double[] test1={-4.3,2.5};
4      char[] test2={'a','f','y'};
```

```
5       for(int i=0;i<test1.length;i++) {
6         System.out.println("test1["+i+"]="+test1[i]);
7       }
8       for(int i=0;i<test2.length;i++) {
9         System.out.println("test2["+i+"]="+test2[i]);
10      }
11    }
12  }
```

Ce programme affiche :

AFFICHAGE

```
test1[0]=-4.3
test1[1]=2.5
test2[0]=a
test2[1]=f
test2[2]=y
```

Les valeurs littérales de type tableau sont très utiles pour tester des méthodes travaillant à partir de tableaux. Bien entendu, dans un "vrai" programme, les tableaux manipulés seront en général fournis par l'utilisateur ou obtenu par un calcul.

Malheureusement, les valeurs littérales de type tableau ne peuvent pas être utilisées comme les valeurs littérales d'un type fondamental. En fait, on doit réserver les valeurs littérales de type tableau à l'initialisation des variables du même type. Il n'est donc pas possible d'écrire par exemple `return {2,3};`, même si la méthode dans laquelle on écrit cette ligne a un résultat de type `int []`, comme l'illustre l'exemple suivant :

Exemple 8.6 :

On considère le programme suivant :

NoReturn

```
1   public class NoReturn {
2     public static int[] f() {
3       return {1,2};
4     }
5     public static void main(String[] args) {
6       int[] k=f();
7     }
8   }
```

Le compilateur refuse le programme et affiche le message d'erreur suivant :

ERREUR DE COMPILATION

```
NoReturn.java:3: illegal start of expression
    return {1,2};
           ^
1 error
```

Il existe cependant un moyen de contourner cette limitation. En effet, il est possible de combiner la création d'un tableau (avec `new`) et l'initialisation de celui-ci. Ainsi peut-on écrire :

```
int[] i=new int[] {2,4,7};
```

Ceci a pour effet de créer un tableau de `int` de longueur 3 et de placer dans ce tableau les éléments indiqués dans la valeur littérale tableau qui suit le `new`. Cette technique peut être utilisée partout où on attend un tableau, sans la limitation évoquée précédemment pour les valeurs littérales. L'exemple suivant illustre les différentes possibilités d'utilisation :

Exemple 8.7 :

On considère le programme suivant :

TableauReturn

```
1  public class TableauReturn {
2    public static void affiche(double[] t) {
3      System.out.println("----------");
4      for(int i=0;i<t.length;i++)
5        System.out.println(i+"->"+t[i]);
6      System.out.println("----------");
7    }
8    public static double[] unTableau() {
9      return new double[] {2.5,3.5,10};
10   }
11   public static double[] unAutreTableau() {
12     double[] d={2.5,3.5,10};
13     return d;
14   }
15   public static void main(String[] args) {
16     double[] t1=unTableau();
17     double[] t2=unTableau();
18     t2[0]=-5;
19     affiche(t1);
20     affiche(t2);
21     double[] t3=unAutreTableau();
22     double[] t4=unAutreTableau();
23     t4[0]=-5;
24     affiche(t3);
25     affiche(t4);
26     affiche(new double[] {1.5,-3.4});
27   }
28 }
```

Il provoque l'affichage suivant :

Affichage

```
----------
0->2.5
1->3.5
2->10.0
----------
----------
0->-5.0
1->3.5
2->10.0
----------
```

```
----------
0->2.5
1->3.5
2->10.0
----------
----------
0->-5.0
1->3.5
2->10.0
----------
----------
0->1.5
1->-3.4
----------
```

Les lignes 9 et 26 montrent qu'on peut utiliser la construction combinant création et initialisation à la place d'un tableau référencé par une variable classique. Les affichages obtenus sont très intéressants. Les deux premiers tableaux affichés (lignes 19 et 20) ne sont pas très surprenant. Comme nous l'avons dit précédemment, la création/initialisation commence par créer un tableau puis le remplit avec les valeurs contenues dans la valeur littérale de type tableau qui suit le `new`. De ce fait les appels des lignes 16 et 17 produisent 2 tableaux distincts (il y a bien deux `new`). Donc quand on modifie le deuxième tableau (ligne 18), ceci n'a pas d'influence sur le premier tableau.

Le point intéressant est que l'utilisation directe d'une valeur littérale (ligne 12 de la méthode `unAutreTableau`) provoque exactement le même résultat, alors qu'il n'y a pas de `new` et qu'on pourrait donc croire que les deux appels à la méthode `unAutreTableau` renvoient le *même* tableau. En fait, il faut comprendre que l'utilisation d'une valeur littérale pour l'initialisation est simplement une abréviation de la construction création/initialisation. La ligne 12 est donc une version simplifiée de :

```
double[] d=new double[] {2.5,3.5,10};
```

REMARQUE

La construction création/initialisation n'existe que depuis la version 1.1 de `Java` et n'est donc pas utilisable dans les versions antérieures.

8.5.3 Affectation, copie et clonage

Comme nous l'avons au chapitre 7, et plus particulièrement à la section 7.3.5, la manipulation par référence impose une interprétation très stricte de l'instruction d'affectation. Quand on écrit `a=b`, la variable `a` reçoit le contenu de la variable `b`. Dans le cas des tableaux, qui sont des objets presque comme les autres, cela signifie que `a` fait référence au même tableau que `b` après l'affectation (nous avons déjà étudié à la section 8.4 les effets de bord que cela peut engendrer). Il est courant de croire que le tableau `b` va être recopié dans le tableau `a`, ce qui est **complètement faux**, comme l'illustre l'exemple suivant.

Exemple 8.8 :

On considère le programme suivant :

PasDeCopie

```
1  public class PasDeCopie {
2    public static void main(String[] args) {
3      int[] a=new int[5];
4      int[] b={1,2,3};
5      a=b;
6      System.out.println(a.length);
7      System.out.println(a[2]);
8      b[2]=5;
9      System.out.println(a[2]);
10   }
11 }
```

Ce programme produit l'affichage suivant :

AFFICHAGE

```
3
3
5
```

Il est donc clair qu'après l'exécution de la ligne 5, la variable `a` fait référence au même tableau que la variable `b`. On peut se demander ce qu'est devenu le tableau de 5 cases créé à la ligne 3. En fait, comme aucune variable ne lui fait référence après l'exécution de la ligne 5, il sera détruit par l'éboueur (cf la section 7.3.8).

Dans la pratique, il est parfois utile de recopier le contenu d'un tableau dans un autre. Pour ce faire, on peut utiliser la méthode `arraycopy` de la classe `System` :

`void arraycopy(Object src,int sp,Object dest,int dp,int length)`

Cette méthode recopie `length` éléments du tableau `src` (à partir de la position `sp` incluse) dans le tableau `dest` (à partir de la position `dp`).

Voici un exemple d'utilisation :

Exemple 8.9 :

On considère le programme suivant :

UneCopie

```
1  public class UneCopie {
2    public static void main(String[] args) {
3      int[] a=new int[5];
4      int[] b={1,2,3};
5      System.arraycopy(b,0,a,1,3);
6      System.out.println(a.length);
7      System.out.println(a[4]);
8      System.out.println(a[2]);
9      b[1]=5;
10     System.out.println(a[2]);
11   }
12
13 }
```

Le programme produit l'affichage suivant :

AFFICHAGE

```
5
0
2
2
```

Cet affichage est très logique. La ligne 5 se contente en effet de recopier les trois éléments de `b` dans `a`. Comme le contenu de la variable `a` ne change pas (elle désigne toujours le même tableau), il est logique que le premier affichage donne 5, à savoir la longueur du tableau `a`. De plus, comme le tableau `b` est recopié dans `a` à partir de la case 1 de `a`, il est normal que le contenu de la case 4 (comme celui de la case 0, d'ailleurs) reste inchangé, à savoir 0 (d'où le deuxième affichage). Enfin, les deux derniers affichages illustrent que le tableau référence par `b` a bien été recopié dans `a` (le contenu de la case 2 de `a` correspond bien à celui de la case 1 de `b`), sans pour autant que `a` et `b` désignent le même tableau : une modification de `b` n'a pas d'effet sur `a`.

REMARQUE

Il faut être extrêmement attentif lors de l'utilisation de la méthode `arraycopy`. Comme nous le verrons dans la section 8.6.6, la copie du tableau ainsi obtenue est une copie *superficielle*, c'est-à-dire que seul le contenant est recopié. Plus précisément, si le tableau contient des objets, ceux-ci ne sont pas recopiés, ce qui peut permettre des effets de bord assez subtils.

La méthode `arraycopy` possède deux défauts. Tout d'abord, elle est relativement "complexe" à utiliser pour les cas simples dans lesquels on souhaite recopier un tableau entier sans décalage dans un autre tableau. De plus, on doit déjà avoir créé un tableau pour pouvoir y recopier le contenu d'un autre.

On peut contourner ces défauts en utilisant une autre technique, le *clonage*. En effet, les objets tableaux possèdent tous une méthode d'instance appelée `clone` dont le but est justement de recopier le tableau appelant.

Voici un exemple d'utilisation de `clone` :

Exemple 8.10 :

On considère le programme :

RecopieClone

```
1 public class RecopieClone {
2   public static void main(String[] args) {
3     int[] u={2,3};
4     int[] v=(int[])u.clone();
5     v[0]=-1;
6     System.out.println(u[0]);
7     System.out.println(v[0]);
8   }
9 }
```

L'affichage produit est le suivant :

AFFICHAGE

```
2
-1
```

En étudiant le programme, on peut faire deux remarques :
- la méthode `clone` ne prend pas de paramètre ;
- l'appel de `clone` doit être précédé par un rappel du type du tableau qu'on souhaite recopier (cloner). Cette mention est obligatoire et le programme ne peut pas être compilé en son absence.

REMARQUES

Comme pour la méthode `arraycopy`, la copie produite par `clone` est superficielle et peut donc engendrer des effets de bord subtils, comme nous le verrons à la section 8.6.6.

Notons de plus que le clonage de tableau n'est possible que depuis la version 1.1 de `Java` et n'est donc pas utilisable dans les versions antérieures. Au contraire, la méthode `arraycopy` existe depuis toujours en `Java`.

8.6 Tableaux multidimensionnels

8.6.1 Introduction

Pour l'instant, nous avons essentiellement vu des tableaux informatiques représentant une ligne de ce qu'on a coutume d'appeler un tableau. Nous avons tout de même utilisé l'appellation *tableau* car c'est la plus classique en informatique. Cependant, la bonne notion est plutôt celle de liste, comme nous l'avons déjà expliqué.

Fort heureusement, il est possible de représenter informatiquement des tableaux usuels, c'est-à-dire un ensemble de cases structuré selon des lignes et des colonnes. Commençons par un exemple simple :

Tableau2D

```
1  public class Tableau2D {
2    public static void main(String[] args) {
3      int[][] tableau=new int[2][3];
4      for(int i=0;i<tableau.length;i++)
5        for(int j=0;j<tableau[i].length;j++)
6          tableau[i][j]=i+j;
7      for(int i=0;i<tableau.length;i++)
8        for(int j=0;j<tableau[i].length;j++)
9          System.out.println("tableau["+i+","+j+"]->"+tableau[i][j]);
10   }
11 }
```

Ce programme provoque l'affichage suivant :

Affichage

```
tableau[0,0]->0
tableau[0,1]->1
tableau[0,2]->2
tableau[1,0]->1
tableau[1,1]->2
tableau[1,2]->3
```

La ligne :

```
int[][] tableau=new int[2][3];
```

		colonne 0	1	2
ligne	0	0	1	2
	1	1	2	3

FIG. 8.7 – Contenu du tableau

déclare une variable `tableau` qui va faire référence à un tableau **à deux entrées**, aussi appelé tableau **à deux dimensions**. La suite de la ligne crée le tableau correspondant, en indiquant qu'il possède deux *lignes* et trois *colonnes*.

REMARQUES

Il ne faut pas se laisser abuser par les appellations ligne et colonne. Il s'agit d'une règle simple choisie arbitrairement. En fait, le tableau possède une première et une deuxième entrée. *A priori*, rien n'empêche d'appeler colonne la première entrée et ligne la seconde. La seule justification à l'appellation choisie est d'origine mathématique : quand on décrit les éléments d'une matrice, on donne toujours par convention le numéro de ligne en premier.

Il est aussi très important de ne pas confondre dimension d'un tableau et taille(s) d'un tableau. Un tableau à deux dimensions peut très bien avoir 10 lignes !

Les deux boucles emboîtées qui suivent se chargent de remplir le tableau. On remarque que l'utilisation d'une case du tableau nécessite maintenant deux valeurs entières : il y a bien deux entrées dans le tableau, ou deux indices, ou encore un numéro de ligne et un numéro de colonne. La figure 8.7 donne une représentation simple du contenu du tableau après la boucle. On remarque que les deux boucles utilisent `length` pour déterminer l'extension de chacune des dimensions du tableau. `tableau.length` donne toujours la première dimension, c'est-à-dire (par convention) le nombre de lignes. Donc, dans cet exemple, la valeur de `tableau.length` est de 2. Pour obtenir le nombre de colonnes, on utilise `tableau[i].length`. En effet, `tableau[i]` est lui-même un tableau, qui correspond (par convention) à la ligne de numéro `i` du tableau de départ. Pour obtenir le nombre de colonnes, il suffit de déterminer le nombre d'éléments contenus dans une ligne.

L'utilisation élémentaire d'un tableau à deux dimensions ne pose donc pas vraiment de problème. Par contre, nous allons voir que la compréhension exacte de la façon dont ces tableaux sont gérés par l'ordinateur est plus délicate.

8.6.2 Interprétation

Nous avons déjà dit que l'ordinateur abstrait peut manipuler des tableaux de n'importe quel objet. Il est donc possible d'avoir des tableaux de tableaux. Quand on écrit `int [][] t ;`, l'ordinateur comprend donc que `t` désigne un objet tableau dont chaque case contient un objet de type `int []`, c'est-à-dire un tableau d'entiers. Plus précisément, chaque case du premier tableau contient une **référence** vers un tableau d'entiers.

Si `t` est correctement initialisé, on peut par exemple accéder à `t[0]`, qui est un `int []`, c'est-à-dire un tableau d'entiers. Si on dispose d'une variable `u` de type `int []`, on peut écrire `u=t[0] ;`, puis ensuite accéder aux éléments de `u`, soit par exemple `u[1]` (si cet élément existe). De plus, cette opération est directement possible au niveau de `t`, ce qui autorise à écrire `t[0][1]`.

De ce fait, la notion de tableau à deux entrées est un cas particulier de celle de tableau de tableaux. La seule "astuce" permise par `Java` est de créer d'un seul coup un tableau à deux dimensions avec un unique `new`. Toutes les autres opérations permises sur les tableaux à deux dimensions se déduisent très simplement de la définition même des tableaux.

Prenons par exemple un tableau simple créé par le code suivant :

Tableau

```
1 int[][] tab=new int[2][3];
2 for(int i=0;i<2;i++)
3   for(int j=0;j<3;j++)
4     tab[i][j]=i+j+1;
```

La figure 8.8 donne une représentation graphique de la mémoire après l'exécution du morceau de programme qui précède. On voit en particulier que le premier objet tableau contient en fait des références vers d'autres objets correspondants à ses lignes.

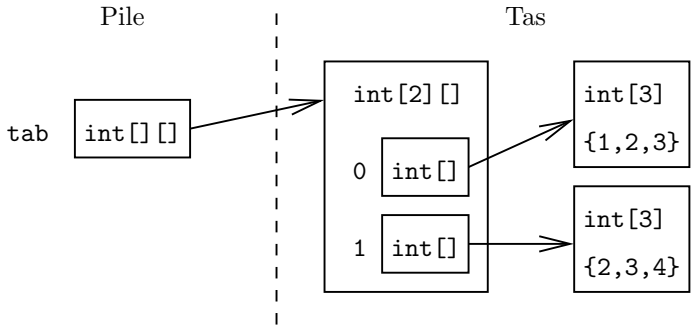

FIG. 8.8 – Tableau à deux dimensions en mémoire

REMARQUE

La représentation de la figure 8.8 montre qu'il n'est plus vraiment possible d'utiliser des valeurs littérales pour la représentation des tableaux à deux dimensions.

8.6.3 Cas général

De façon générale, une variable faisant référence à tableau à deux entrées contenant des éléments de type *T* sera déclarée par :

T`[][]` *identificateur*`;`

La création d'un objet de ce type et le placement d'une référence sur le résultat dans une variable appropriée se fait en écrivant :

identificateur `= new` *T*`[n][p];`

Dans cette instruction, `n` désigne par convention le **nombre de lignes** du tableau et `p` **nombre de colonnes**. Le tableau créé peut donc contenir `np` éléments.
Pour accéder à un élément donné, on utilise :

identificateur`[i][j]`

où `i` et `j` sont des entiers respectivement compris entre `0` et `n-1` et entre `0` et `p-1`.
Le nombre de lignes du tableau s'obtient en utilisant la notation

identificateur`.length`

Pour obtenir le nombre de colonnes de la ligne `i` (pour `i` compris entre `0` et `n-1`), on utilise :

identificateur`[i].length`

Si le tableau a été créé comme indiqué précédemment, la valeur obtenue est toujours égale à `p`, pour toute valeur acceptable de `i`.

REMARQUE

En ajoutant ainsi les paires de crochets les unes après les autres, on augmente le nombre d'entrées (c'est-à-dire le nombre de dimensions) du tableau manipulé. On peut ainsi manipuler un tableau à 3 dimensions en écrivant `int[][][]`.

8.6.4 Valeur littérale

Comme nous l'avons déjà dit, un tableau à deux dimensions est en fait un tableau de tableaux. De ce fait, il est facile d'utiliser des valeurs littérales de type tableau à deux dimensions : il suffit d'écrire une valeur littérale de type tableau dont les éléments sont eux-mêmes des tableaux.

Exemple 8.11 :

Le programme suivant crée un tableau de réels à deux dimensions en utilisant une valeur littérale :

TableauValeur2D

```
1  public class TableauValeur2D {
2    public static void main(String[] args) {
3      double[][] tab={{-4.3,2.5},
4                      {2.5,5.4},
5                      {3.2,6.7}};
6      for(int i=0;i<tab.length;i++)
7        for(int j=0;j<tab[i].length;j++)
8          System.out.println("tab["+i+"]["+j+"]="+tab[i][j]);
9    }
10 }
```

L'affichage obtenu est le suivant :

AFFICHAGE

```
tab[0][0]=-4.3
tab[0][1]=2.5
tab[1][0]=2.5
tab[1][1]=5.4
tab[2][0]=3.2
tab[2][1]=6.7
```

On remarque que la notation utilisée dans le programme permet de lire facilement la valeur du tableau à partir de la valeur littérale. Ceci étant, il s'agit simplement d'une convention d'écriture et rien n'empêche de tout placer sur une seule ligne.

8.6.5 Construction incomplète et tableaux irréguliers

Comme nous l'avons déjà indiqué précédemment, un tableau à deux entrées est avant tout un tableau de tableaux, ce qui permet des constructions assez complexes (et malheureusement, des pièges subtils, comme nous le verrons dans la section 8.6.6).

Exemple de tableau irrégulier

Comme nous venons de le voir, l'opérateur **new** permet de créer directement un tableau à deux dimensions. Cependant, ce n'est absolument pas obligatoire : il est possible de ne créer que la

"première couche" du tableau. En effet, imaginons par exemple une variable `t` de type `int [][]`. Alors, `t` doit contenir une référence vers un tableau donc chaque case doit elle-même contenir une référence de type `int[]` (c'est-à-dire une référence désignant un tableau d'entier). Rien n'oblige à créer *d'un seul coup* le tableau désigné par `t` *et* les tableaux désignés par ce dernier.

On peut en effet créer un tableau de type `int[][]` par l'instruction suivante : `t=new int[2][];`. Ceci a pour conséquence de placer dans le tas un tableau à deux cases contenant chacune une référence `null`. Chaque case est prévue pour recevoir une référence sur un tableau de type `int []`. On peut donc ensuite écrire `t[0]=new int[2];` et `t[1]=new int[3];`. De ce fait, la première ligne de `t` possède 2 cases alors que la seconde ligne possède trois cases. On obtient ainsi la valeur 2 pour `t[0].length` et la valeur 3 pour `t[1].length`.

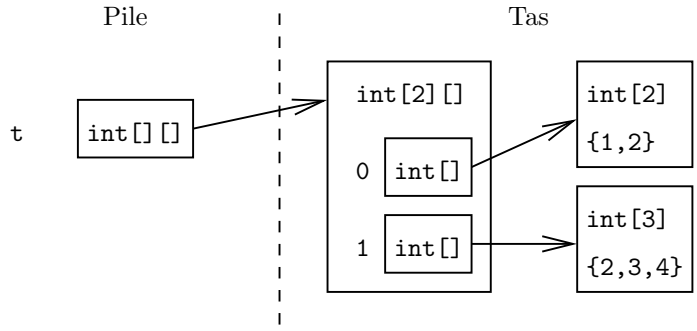

FIG. 8.9 – Tableau à deux dimensions irrégulier en mémoire

Cet exemple montre que les lignes d'un tableau sont complètement indépendantes les unes des autres en `Java`, ce qui peut parfois poser des problèmes d'interprétation, d'autant plus que la situation décrite précédemment peut s'obtenir de façon très simple avec une valeur littérale :

Tableau2D

```
1 int[][] t={{1,2},
2            {2,3,4}};
```

Dans la mémoire de l'ordinateur, cette initialisation produit la configuration illustrée par la figure 8.9.

Cas général d'allocation incomplète

Quand on manipule un tableau à plusieurs dimensions, de type `T[]...[]`, on peut utiliser une forme incomplète de l'opérateur `new` : au lieu de préciser la taille de chaque dimension (par exemple le nombre de lignes, puis le nombre de colonnes dans le cas à deux dimensions), on peut laisser indéterminées les dernières dimensions. On doit cependant :

1. donner la taille de la première dimension ;
2. laisser indéterminées *toutes* les dimensions à partir d'un certain rang : on ne peut pas, par exemple, fixer la troisième dimension et laisser indéterminée la deuxième ;
3. pour chaque dimension indéterminée, écrire une paire de crochets sans contenu.

Exemple 8.12 :

Voici quelques constructions valides :

```
int[][][] t=new int[2][3][];
double[][][] t=new double[1][][];
int[][] t=new int[1][];
```

Voici des constructions interdites :

```
int[][][] t=new int[2][][3];
double[][][] t=new double[][1][2];
int[][] t=new int[][];
```

Pour chaque dimension déterminée, l'ordinateur effectue la création du tableau correspondant. De plus, les références sont correctement positionnées : si on fixe par exemple les deux premières dimensions d'un tableau, chaque case du tableau des lignes contient une référence vers un tableau représentant cette ligne. Dès qu'une dimension est indéterminée, la création des tableaux se termine, et les cases du dernier tableau créé contiennent toutes la référence **null**.

Exemple 8.13 :

Analysons les constructions valides de l'exemple précédent :

- `int[][][] t=new int[2][3][] ;`
  Pour cette création, l'ordinateur commence par la construction un objet tableau de type `int[][][]` contenant 2 cases. Il place la référence correspondante dans `t`. Ensuite, dans chaque case de cet objet, il place une référence vers un objet tableau de type `int[][]` contenant 3 cases. Comme la troisième dimension est indéterminée, les cases de ces tableaux contiennent toutes `null`. L'état de la mémoire est illustré par la figure 8.10.
- `double[][][] t=new double[1][][] ;`
  Pour cette création, seul l'objet tableau de type `double[][][]` est créé, car les deux dernières dimensions sont indéterminées. Après la création, la variable `t` contient donc une référence vers un tableau à 1 case, contenant la référence `null`. L'état de la mémoire est illustré par la figure 8.11.
- `int[][] t=new int[1][] ;`
  Pour cette création, seul l'objet tableau de type `int[][]` est créé, car la dernière dimension est indéterminée. Après la création, la variable `t` contient donc une référence vers un tableau à 1 case, contenant la référence `null`. L'état de la mémoire est illustré par la figure 8.12.

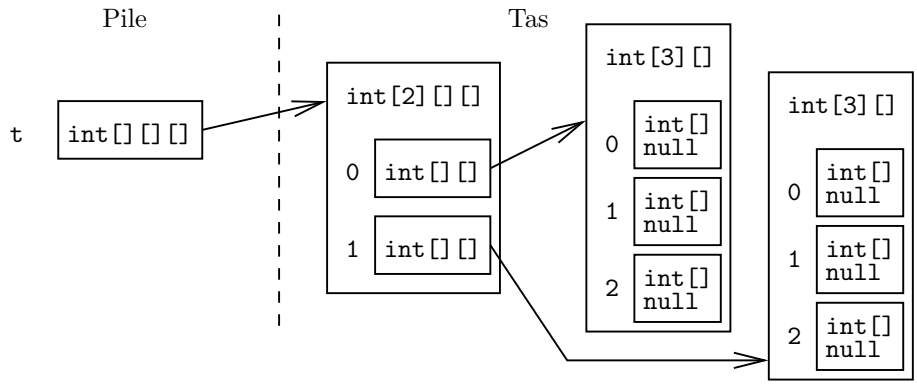

FIG. 8.10 – Tableau à trois dimensions irrégulier en mémoire

Manipulation des "lignes"

L'allocation incomplète permet de différer la création des tableaux contenus dans le tableau principal. Dans le cas à deux dimensions, on peut de ce fait choisir le nombre de lignes du tableau, puis créer, quand le besoin s'en fait sentir, les tableaux qui vont représenter les lignes en question. De façon générale, le contenu d'un tableau peut être modifié comme bon nous semble, à partir du

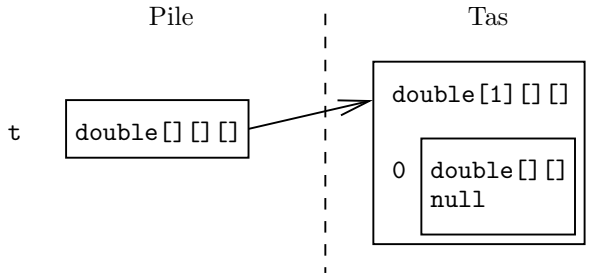

FIG. 8.11 – Un autre tableau à trois dimensions irrégulier en mémoire

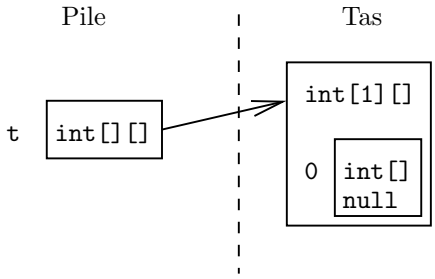

FIG. 8.12 – Un autre tableau à deux dimensions irrégulier en mémoire

moment où les types sont respectés. Or, un tableau de type *T*`[][]` peut être considéré comme un tableau à deux dimensions de donc chaque case contient une référence ou une valeur de type *T*, mais aussi comme un tableau à une dimension donc chaque case contient une référence de type *T*`[]`. On peut donc manipuler le tableau élément par élément, mais aussi ligne par ligne, en particulier au moment de l'allocation. Ceci reste vrai quand le tableau comporte plus de dimensions, comme l'illustre l'exemple suivant :

Exemple 8.14 :

Reprenons l'exemple de création suivant :

```
double[][][] t=new double[2][][];
```

Dans ce cas, les cases `t[0]` et `t[1]` contiennent `null` et sont destinées à recevoir des références vers des tableaux de type `double[][]`. On peut donc écrire :

```
t[0]=new double[2][3];
```

ce qui alloue un tableau à deux dimensions et place la référence correspondante dans la case `t[0]`. On peut ensuite écrire :

```
t[1]=new double[2][];
```

Dans ce cas, `t[1]` reçoit une référence vers un tableau de type `double[][]` contenant deux cases. Chaque case contient `null`. On peut enfin terminer l'allocation par :

```
t[1][0]=new double[2];
t[1][1]=new double[3];
```

La figure 8.13 donne l'état de la mémoire après l'ensemble des allocations.

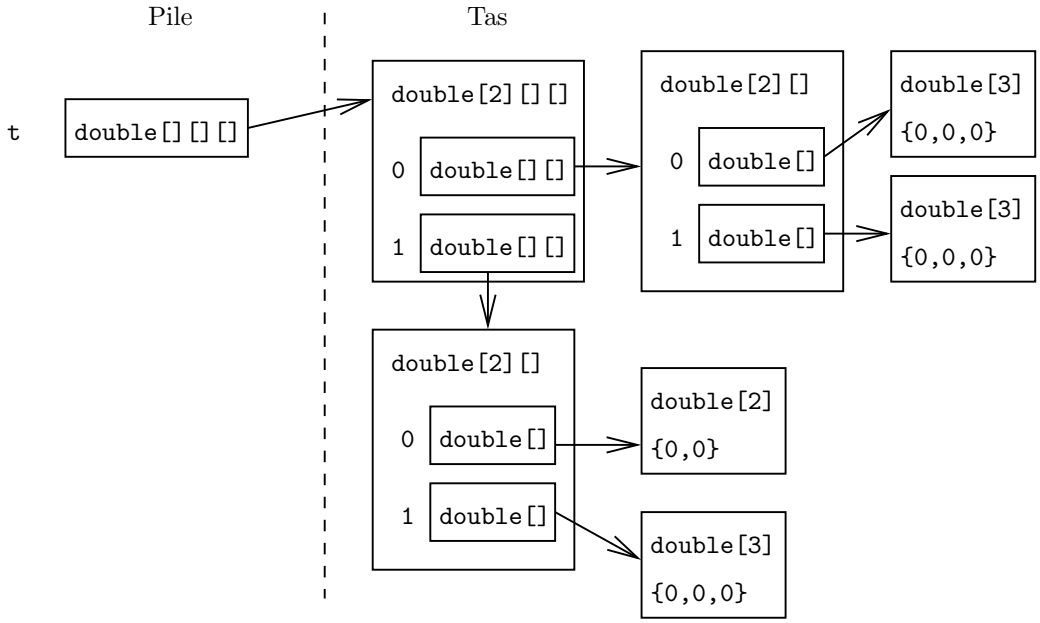

FIG. 8.13 – Un autre tableau à trois dimensions irrégulier en mémoire

REMARQUE

Il faut donc toujours garder à l'esprit que pour tout type *T*, un tableau de type *T*[] contient des cases et que chaque case se comporte comme une variable de type *T*. Ceci reste valable même si *T* correspond lui-même à un type tableau, comme l'a illustré la présente section.

8.6.6 Quelques pièges

Il est important de bien comprendre que la notion de tableau de tableaux est fondamentalement différente de celle de tableau simple. Nous allons donner dans cette section quelques exemples des pièges les plus classiques que posent les tableaux à deux entrées.

Effet de bord

Nous avons vu dans la section 8.4 qu'il était impossible de changer la taille d'un tableau par effet de bord. Cela reste bien sûr vrai pour un tableau à deux dimensions, **mais seulement pour son nombre de lignes**, comme le montre programme suivant :

BordTableau2D

```
1  public class BordTableau2D {
2    public static void main(String[] args) {
3      int[][] a={{2,5},
4                 {3,4}};
5      int[][] b=a;
6      b[0]=new int[1];
7      b[0][0]=-1;
8      System.out.println(a[0].length);
9      System.out.println(a[0][0]);
10   }
11 }
```

Ce programme affiche :

AFFICHAGE

```
1
-1
```

En effet, `b` désigne le même tableau que `a`. De ce fait, on peut changer le contenu de `a` par l'intermédiaire de `b` (mais on ne peut pas changer la longueur de `a`, c'est-à-dire le nombre de lignes du tableau). Or, le contenu de `a` se résume à des *références* sur des tableaux d'entiers. Il est donc facile d'utiliser `b` pour changer une de ces références, c'est-à-dire le tableau *entier* qui représente une ligne de `a`. Dans le programme, on utilise cette possibilité pour remplacer la première ligne de `a` par une nouvelle ligne réduite à un seul élément dont la valeur n'est même plus celle du premier élément de l'ancienne ligne de `a`. La figure 8.14 donne l'état de la mémoire à la fin du programme. La flèche en pointillés correspond à la valeur initiale de la référence contenue dans `a[0]`, avant l'exécution de la ligne `b[0]=new int[1] ;`.

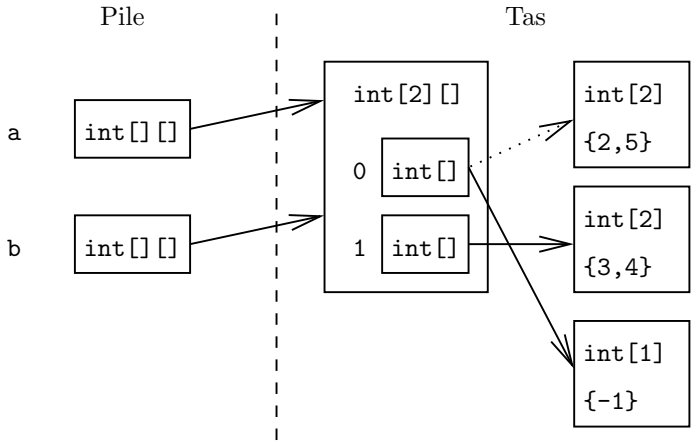

FIG. 8.14 – Tableau à deux dimensions et effet de bord

De la même façon, on peut placer le contenu d'une ligne de `a`, par exemple `a[0]`, dans une variable `c` de type `int[]`. On va ainsi pouvoir modifier le contenu d'une ligne de `a` en passant par la variable `c` qui n'est même pas du même type que `a` ! Comme on le constate, les effets de bord deviennent à ce niveau un peu délicats à appréhender. Il faut cependant se rassurer : tout ceci est très logique et demande simplement une rigueur importante dans l'analyse d'un programme.

Notons pour finir cette section que les effets de bord peuvent devenir encore plus complexe quand deux lignes d'un tableau sont identiques. Dans l'exemple précédent, il est parfaitement possible d'ajouter une instruction de le forme `a[0]=a[1] ;`, ce qui a pour effet d'indiquer que les deux cases du tableau `a` font maintenant référence au *même* tableau d'entiers. On peut donc avoir un effet de bord au sein même d'un tableau !

Clonage

Comme nous l'avons évoqué à la section 8.5.3, il faut bien comprendre que la méthode `clone` des tableaux se contente de recopier le contenu *direct* du tableau et que si ce tableau contient des objets, ceux-ci ne sont pas recopiés (toute la discussion qui va suivre s'applique exactement de la même façon à la méthode `arraycopy`). Étudions alors l'exemple suivant :

RecopieClone2D

```
1  public class RecopieClone2D {
2    public static void main(String[] args) {
3      int[][] u={{2,3},
4                 {5,6}};
5      int[][] v=(int[][])u.clone();
6      v[0]=new int[1];
7      v[0][0]=-2;
8      v[1][1]=3;
9      System.out.println(u[0].length);
10     System.out.println(u[0][1]);
11     System.out.println(u[1][1]);
12   }
13 }
```

L'affichage produit par ce programme est le suivant :

AFFICHAGE

```
2
3
3
```

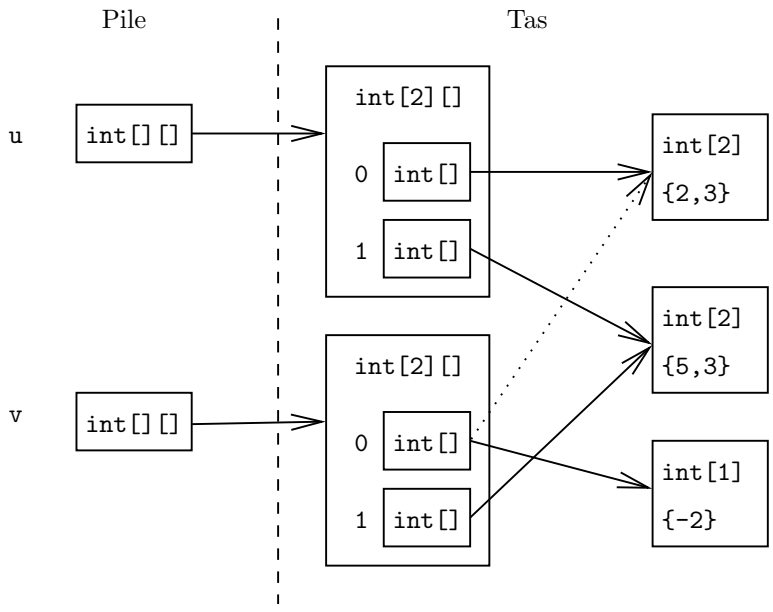

FIG. 8.15 – Tableau à deux dimensions et clonage

Les deux premières lignes ne sont pas des surprises, mais par contre, la dernière n'est pas compatible avec une vision naïve du clonage. En effet, comme `v` contient une copie de `u`, on ne devrait pas pouvoir modifier `u` en se servant de `v`. En fait, c'est parfaitement exact, car l'instruction `v[1][1]=3 ;` ne modifie pas `v`, elle modifie plus précisément `v[1]`, c'est-à-dire la deuxième ligne de `v`, qui est un tableau d'entiers. Or, le clonage porte sur `v`, pas sur son contenu. Donc, le tableau `v[1]` est le même que le tableau `u[1]`, ce qui explique la "transmission" de la modification. La figure 8.15

montre l'état de la mémoire à la fin du programme, la flèche en pointillés désignant la première valeur de `v[0]`, avant l'exécution de l'instruction `v[0]=new int[1] ;`.

Le seul moyen d'obtenir une copie complète est de cloner chacune des lignes contenues dans `v`, ce qui s'obtient par une simple boucle.

REMARQUE

Tout le contenu de cette section s'applique de façon générale **aux tableaux d'objets**, plus particulièrement si les objets en question sont modifiables. On peut donc observer des comportements très "étranges" avec des tableaux de type `StringBuffer[]`, par exemple, comme l'illustre la section suivante.

8.7 Tableaux d'objets

8.7.1 Introduction

Comme nous avons étudié avec précision les tableaux à deux dimensions et plus, les tableaux d'objets deviennent presque une formalité. En effet, un tableau à deux dimensions est un tableau de tableaux, donc un tableau d'objets. Or, la principale difficulté des tableaux d'objets réside dans les effets de bord que nous venons d'étudier dans le cas particulier des tableaux de tableaux...

Il faut d'abord noter que `Java` n'impose pas de limitation aux tableaux d'objets : pour tout type objet *T*, on peut manipuler des tableaux de type *T*`[]`. Il est bien sûr possible de manipuler des tableaux à plusieurs dimensions contenant des objets. Les deux principales difficultés liées à la manipulation de tableaux d'objets sont la création et les effets de bord.

8.7.2 Création

Considérons un exemple simple :

TableauString

```
1  public class TableauString {
2    public static void display(String[] t) {
3      for(int i=0;i<t.length;i++)
4        System.out.println(t[i]);
5    }
6    public static void main(String[] args) {
7      String[] tab;
8      tab = new String[3];
9      display(tab);
10     tab[0]="ABC";
11     tab[1]="DEF";
12     tab[2]="HIJ";
13     display(tab);
14   }
15 }
```

Ce programme affiche :

AFFICHAGE

```
null
null
```

```
null
ABC
DEF
HIJ
```

Ce comportement est parfaitement cohérent avec les règles énoncées à la section 8.5.1. En effet, cette section indique que pour un tableau d'objets, chaque case du tableau est initialisé à la valeur **`null`**. Cette valeur indique en fait que la case du tableau ne fait référence à aucun objet.

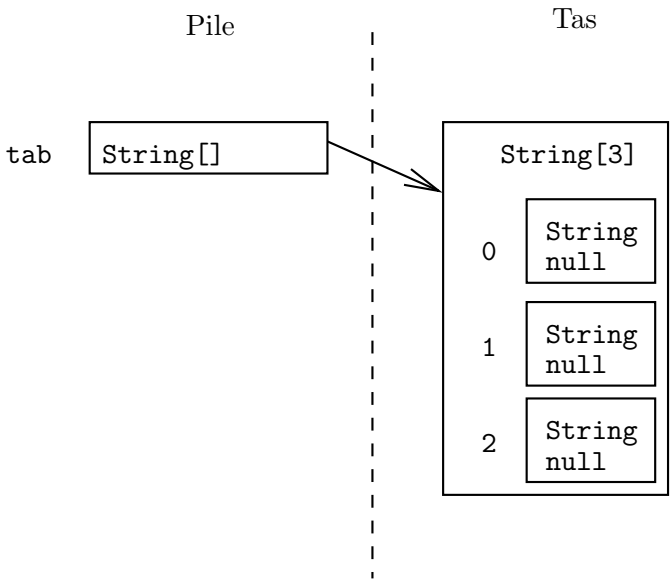

FIG. 8.16 – Création d'un tableau d'objets : première phase

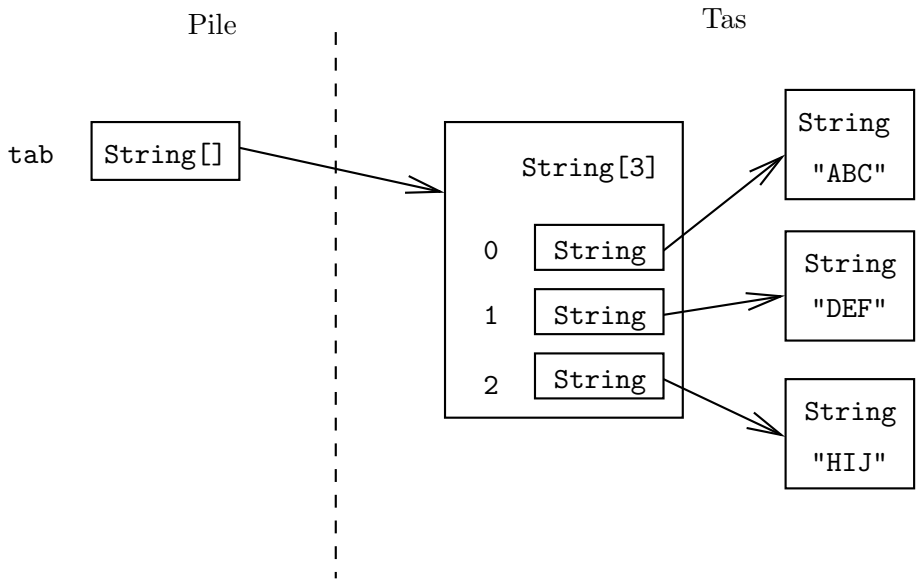

FIG. 8.17 – Création d'un tableau d'objets : seconde phase

De ce fait, la création d'un tableau d'objets se fait nécessairement en deux temps. Dans un premier temps, on fabrique le tableau proprement dit. Après la création, le tableau est "vide". Cela signifie simplement que chaque case contient la référence **null** et ne désigne donc aucun objet (cf la figure 8.16 qui illustre cette première phase). Dans un deuxième temps, on peut fixer ne contenu de chaque case du tableau, ce qui signifie placer dans chaque case une référence vers un objet du type considéré. Dans l'exemple étudié, on place simplement une chaîne de caractères, ce qui ne nécessite pas de **new** (cf la figure 8.17 qui illustre cette seconde phase). Cependant, dans le cas général, on doit utiliser un `new`, comme l'illustre l'exemple suivant :

TableauStringBuffer

```
1  public class TableauStringBuffer {
2    public static void display(StringBuffer[] t) {
3      for(int i=0;i<t.length;i++)
4        System.out.println(t[i]);
5    }
6    public static void main(String[] args) {
7      StringBuffer[] tab;
8      tab = new StringBuffer[3];
9      display(tab);
10     tab[0]=new StringBuffer("ABC");
11     tab[1]=new StringBuffer("DEF");
12     tab[2]=new StringBuffer("HIJ");
13     display(tab);
14   }
15 }
```

Ce programme est exactement similaire au précédent (et produit le même affichage), sauf qu'il utilise des `StringBuffer`s à la place des `String`s.

REMARQUES

Dans la pratique, rien n'oblige à remplir le tableau d'un seul coup. De plus, il est parfois utile de laisser certaines cases vides (c'est-à-dire contenant la référence `null`), pour indiquer que le résultat correspondant n'est pas encore connu par exemple.

Notons qu'il n'existe pas d'allocation groupée pour les tableaux d'objets, au contraire des tableaux de tableaux.

Grâce aux valeurs littérales de type tableau, on peut cependant obtenir une allocation en un seul temps, comme l'illustre le programme suivant :

TableauStringBuffer2

```
1  public class TableauStringBuffer2 {
2    public static void display(StringBuffer[] t) {
3      for(int i=0;i<t.length;i++)
4        System.out.println(t[i]);
5    }
6    public static void main(String[] args) {
7      StringBuffer[] tab = { new StringBuffer("ABC"),
8                             new StringBuffer("DEF"),
9                             new StringBuffer("HIJ") };
10     display(tab);
11   }
12 }
```

Comme nous l'avons vu à la section 8.5.2, une valeur littérale de type tableau s'obtient en donnant la liste des valeurs qu'on souhaite placer dans le tableau, séparées par des virgules et entourées d'une paire d'accolades. Dans le cas des tableaux d'objets, la situation est exactement similaire, sauf que la production d'objets implique l'utilisation de **new**. Le cas des `String`s est plus simple et on pourrait écrire :

TableauString2

```
1 String[] tab = { "ABC","DEF","HIJ" };
```

8.7.3 Effets de bord

Le principal problème lié à l'utilisation des tableaux d'objets est celui des effets de bord. Nous savons en effet que la manipulation des objets par référence peut poser quelques problèmes : on peut modifier un objet en utilisant une variable et observer les effets de cette modification grâce à un autre variable. Or, un tableau ne contient pas directement des objets mais des références. On peut donc observer des résultats inattendus, comme l'illustre l'exemple suivant :

Exemple 8.15 :

On considère le programme :

TableauObjetBord1

```
1  public class TableauObjetBord1 {
2    public static void display(StringBuffer[] t) {
3      for(int i=0;i<t.length;i++)
4        System.out.println(t[i]);
5    }
6    public static void main(String[] args) {
7      StringBuffer[] tab = { new StringBuffer("ABC"),
8                             new StringBuffer("DEF"),
9                             new StringBuffer("HIJ") };
10     StringBuffer s=tab[0]; // préparation de l'effet de bord !
11     s.append(" SUITE"); // effet de bord
12     display(tab);
13   }
14 }
```

Celui-ci affiche :

AFFICHAGE

```
ABC SUITE
DEF
HIJ
```

On voit donc que le "contenu" du tableau est modifié par effet de bord grâce à la variable `s`. En fait, il faut bien comprendre que le tableau ne contient que des références et qu'il est donc parfaitement normal qu'il apparaisse comme modifié quand un des objets auquel il fait référence est modifié. La figure 8.18 représente l'état de la mémoire juste avant l'appel `s.append(" SUITE")` et montre bien pourquoi cet appel provoque un effet de bord.

Bien entendu, les effets de bord ne sont pas limités à une utilisation directe et peuvent apparaître dans les méthodes, comme le montre l'exemple suivant.

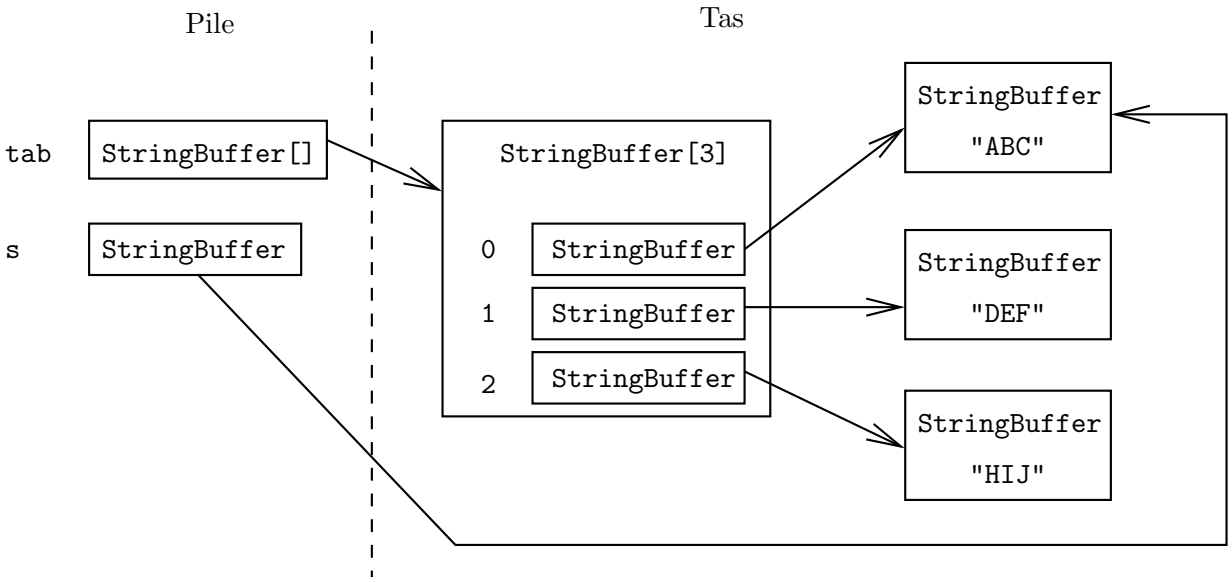

FIG. 8.18 – Tableau d'objets et effet de bord

Exemple 8.16 :

On considère le programme suivant :

TableauObjetBord2

```
1  public class TableauObjetBord2 {
2    public static void effetNiveauTableau(StringBuffer[] t) {
3      t[0]=new StringBuffer("UVW");
4    }
5    public static void effetNiveauObjets(StringBuffer[] t) {
6      t[1].append("test");
7    }
8    public static void main(String[] args) {
9      StringBuffer[] tab = { new StringBuffer("ABC"),
10                            new StringBuffer("DEF"),
11                            new StringBuffer("HIJ") };
12     effetNiveauTableau(tab);
13     System.out.println(tab[0]);
14     effetNiveauObjets(tab);
15     System.out.println(tab[1]);
16   }
17 }
```

Comme leur noms l'indiquent, la méthode `effetNiveauTableau` réalise un effet de bord classique au niveau du tableau (elle change le contenu d'une case) alors que `effetNiveauObjets` modifie l'objet auquel une case du tableau fait référence, réalisant de ce fait un effet de bord plus subtil.

8.8 Conseils d'apprentissage

Les tableaux sont un outil incontournable en `Java`. Il est presque impossible de réaliser un programme complet sans utiliser de tableaux. Or, les tableaux mélangent plusieurs difficultés :

- les tableaux sont des objets, ce qui signifie une manipulation plus délicate que pour les types fondamentaux, comme nous l'avons longuement expliqué dans le chapitre 7 et dans le présent chapitre ;
- les tableaux introduisent de nouvelles constructions syntaxiques et de nouvelles instructions (clonage, construction partielle, etc.) ;
- comme les `String`s et les `StringBuffer`s, les tableaux posent de nouveaux problèmes algorithmiques (que nous n'avons pas abordés dans ce chapitre).

Voici quelques conseils pour acquérir une bonne maîtrise des tableaux :

- Il faut bien entendu commencer par comprendre et retenir les nouvelles **constructions propres aux tableaux** : type, constructeur, accès au contenu, **longueur**, etc.
- Quand la syntaxe et la sémantique sont acquises, il faut reprendre des algorithmes classiques pour les appliquer aux tableaux : recherche d'un élément dans un tableau, comptage, etc. Ce travail permet de mieux se familiariser avec la manipulation des tableaux.
- Quand les tableaux ne posent plus de problème en tant qu'outils, il est important de revenir sur leur nature. En tant qu'objets, ils posent des problèmes spécifiques d'effet de bord par exemple. L'étude des tableaux en tant qu'objets est d'ailleurs une bonne occasion pour vérifier qu'on maîtrise bien les spécificités des objets (manipulation par référence, gestion mémoire, etc.).
- L'étude des **tableaux à plusieurs dimensions et des tableaux d'objets** est à réserver pour une seconde phase. Il est illusoire de vouloir comprendre les subtilités de ces constructions sans posséder une bonne maîtrise des tableaux simples.

Les tableaux sont un outil puissant, mais leur apprentissage reste difficile. Il est donc parfaitement normal de devoir leur consacrer un volume de travail important.

CHAPITRE 9

Création d'objets

Sommaire

Introduction

Le but de la programmation présentée dans ce cours est avant tout la traduction informatique d'une modélisation mathématique du réel. Il s'agit par exemple de programmer un modèle statistique afin d'en estimer numériquement les paramètres : ceci permettra d'effectuer une prédiction ou une analyse de la réalité représentée par ce modèle. De façon plus générale, l'un des buts de l'informatique est de représenter le réel, éventuellement sans passer par un cadre mathématique.

Pour que cette représentation soit efficace et utile, les langages de programmation modernes (comme `Java`) permettent de définir des *objets*. Un objet est la représentation informatique d'une partie (abstraite ou concrète) de la réalité. On peut par exemple définir un objet *voiture* qui va décrire informatiquement une voiture : on peut adopter un point de vue abstrait (celui du code de la route) pour lequel une voiture possède simplement trois pédales (accélérateur, frein et embrayage), un levier de vitesses et un volant ; on peut aussi adopter un point de vue très concret (celui du garagiste) pour lequel une voiture est un ensemble de pièces possédant chacune une référence précise.

Se pose alors le problème de la représentation : nous savons pour l'instant manipuler des valeurs numériques (entières et réelles), des caractères et du texte ; nous savons aussi créer des tableaux de ces valeurs. Il semble alors délicat de construire une représentation informatique agréable (simple à utiliser), même pour des "réalités" très mathématiques. Prenons l'exemple des nombres complexes. L'idéal serait d'avoir un type `Complexe` permettant de manipuler simplement de tels nombres. Or, pour connaître la valeur d'un nombre complexe, il faut 2 réels. On peut donc choisir de représenter les éléments de $\mathbb{C}$ comme un tableau de réels (`double[]`), mais la manipulation reste difficile. En effet, il n'est pas facile de faire la différence entre un tableau de réels quelconque et un tableau *représentant* un complexe. De plus, la définition d'opérations sur ces complexes (par exemple la somme) passe par la définition de méthodes de classe, comme par exemple une méthode `somme` dans

la classe `Complexe`. On devra donc écrire des opérations du genre `a=Complexe.somme(b,c) ;`, ce qui devient vite fastidieux. En outre, ces méthodes doivent impérativement vérifier que les tableaux de réels qu'elles reçoivent en paramètre contiennent bien exactement deux réels, ce qui les rend plus lentes.

Si on compare cette tentative avec les objets `String`, on constate que l'abstraction proposée par ces derniers est nettement plus satisfaisante. On ne sait pas *comment* les objets `String` sont représentés. Par contre, on dispose de méthodes d'instance (plus agréables à utiliser) permettant de les manipuler aisément. On pourrait bien sûr représenter les `String` par des tableaux de `char`, sous la forme `char []`. On aurait alors les mêmes problèmes que pour la proposition de `Complexe`, en particulier en ce qui concerne l'écriture de méthodes. De plus, on perdrait un avantage indéniable des objets `String` : ceux-ci ne sont pas modifiables et permettent donc d'écrire un programme avec des objets exactement comme si on manipulait des types fondamentaux (voir la section 7.4.6).

On devine grâce à cette introduction que la *création* de nouveaux objets va être la technique idéale pour introduire des représentations informatiques efficaces de la réalité. Le but de ce chapitre est d'expliquer comment on peut créer ses propres objets en langage `Java`

9.1 Les objets comme dispositif de stockage

9.1.1 Exemple introductif : moyenne et variance

Considérons l'exemple très élémentaire du calcul de la moyenne et de la variance d'un ensemble de valeurs numériques. L'ensemble de valeurs sera bien entendu représenté par un tableau de réels (un `double[]`). Le problème posé par l'écriture du programme est celui du stockage de la moyenne et de la variance. On peut bien sûr utiliser un tableau de deux réels, le premier représentant par exemple la moyenne, et le second la variance. Le principal défaut de cette approche est que la convention utilisée n'est pas très explicite : pourquoi placer la moyenne en premier et pas la variance ? Et surtout, comment se souvenir de cette convention ? Pour finir, le code de la méthode en elle-même n'est pas très lisible, comme le montre le programme suivant :

MoyenneVariance

```
1  public double[] moyenneVariance(double[] t) {
2    double[] result=new double[2];
3    for(int i=0;i<t.length;i++)
4      result[0]+=t[i];
5    result[0]/=t.length;
6    for(int i=0;i<t.length;i++)
7      result[1]+=(t[i]-result[0])*(t[i]-result[0]);
8    result[1]/=t.length;
9    return result;
10 }
```

Grâce aux objets, il est possible de proposer une solution presque aussi simple de programmation, mais plus satisfaisante, car plus explicite. La définition d'objets passe par l'écriture d'une classe. On propose ici la classe suivante :

Statistiques

```
1  public class Statistiques {
2    public double moyenne;
3    public double variance;
4  }
```

Cette définition comporte des éléments familiers, mais dans un contexte nouveau : les lignes 2 et 3 contiennent en effet des déclarations de variables. Contrairement à l'habitude, ces déclarations apparaissent en dehors de toute méthode. De plus, elles sont précédées du mot clé **public**. Pour expliquer le sens de ces déclarations, commençons par un exemple d'utilisation de la classe ainsi créée :

CalculMoyenneVariance

```
1  public class CalculMoyenneVariance {
2    public static Statistiques moyenneVariance(double[] t) {
3      Statistiques result=new Statistiques();
4      for(int i=0;i<t.length;i++)
5        result.moyenne+=t[i];
6      result.moyenne/=t.length;
7      for(int i=0;i<t.length;i++)
8        result.variance+=(t[i]-result.moyenne)*(t[i]-result.moyenne);
9      result.variance/=t.length;
10     return result;
11   }
12   public static void main(String[] args) {
13     Statistiques s=moyenneVariance(new double[] {8,4,-2,6,10});
14     System.out.println(s.moyenne);
15     System.out.println(s.variance);
16   }
17 }
```

Ce programme affiche :

Affichage

```
5.2
16.96
```

Même si les mécanismes mis en jeu peuvent sembler obscurs pour l'instant, il est clair que l'utilisation d'un objet rend le programme plus explicite : on utilise à aucun moment de code pour indiquer qu'on souhaite manipuler la moyenne ou la variance. Tout est explicite et basé sur un principe simple : la moyenne est stockée dans une variable appelée `moyenne`, alors que la variance est stockée dans `variance`. Dans les sections suivantes, nous allons expliquer les différents mécanismes qui interviennent dans la définition d'objet, en nous basant sur l'exemple qui vient d'être présenté.

9.1.2 Toute classe définit un type

Le premier point nouveau de l'exemple précédent s'observe à la ligne 2 de la classe proposée, `CalculMoyenneVariance`. On remarque en effet que `Statistiques` est utilisé comme un type (exactement comme `StringBuffer`, par exemple). Cette utilisation se confirme aux lignes 3 et 13. En fait, nous avons ici une utilisation d'un mécanisme universel : **toute définition de classe engendre une définition de type**. Dans la pratique, c'est souvent complètement inutile, car sans constructions additionnelles, le type correspondant n'est pas vraiment utilisable.

Nous avons vu au chapitre 7, que tout objet possède un type qui est en fait décrit par une classe. Il n'y a rien de particulier à faire pour qu'une classe définisse un type, ce mécanisme est automatique. A chaque fois que nous souhaiterons définir un type objet, nous devrons donc simplement choisir son nom (`Statistiques` dans l'exemple en cours d'étude), puis écrire la classe correspondante.

9.1.3 Les variables d'instance

Les lignes 2 et 3 de la classe `Statistiques` correspondent à des déclarations de variables. Il s'agit de variables particulières appelées **variables d'instance**. Comme nous l'avons dit au chapitre 7, une classe qui permet la définition effective (c'est-à-dire utile) d'objets est la description générique de tous les objets possibles instances de cette classe. Dans notre exemple simple, la classe se contente de décrire le **contenu** de ses instances, c'est-à-dire le contenu des objets du type qu'elle décrit. La signification exacte de la ligne 2 est donc la suivante : **chaque** objet de type `Statistiques` **contient** une variable de type `double` appelée `moyenne`. De la même façon, la ligne 3 indique que **chaque** objet instance de `Statistiques` **contient** une variable de type `double` appelée `variance`[1].

De façon plus générale, la déclaration d'une variable **en dehors** de toute méthode indique que chaque objet instance de la classe qui contient cette définition contiendra la variable indiquée. On peut donc définir par exemple une classe `Complexe` de la façon suivante :

Complexe

```
1 public class Complexe {
2   public double re;
3   public double im;
4 }
```

Tout objet instance de cette classe contiendra donc deux variables de type `double`, appelées `re` et `im` (et représentant donc respectivement les parties réelle et imaginaire du nombre complexe étudié).

La méthode `moyenneVariance` renseigne sur l'utilisation des variables d'instance. Étant donnée une référence `a` vers un objet du type considéré (ici `Statistiques`), on peut manipuler directement les variables d'instance de l'objet en question grâce à la notation "pointée", c'est-à-dire en faisant suivre la référence d'un point et du nom de la variable d'instance choisie. La ligne `5` de la classe `CalculMoyenneVariance` contient ce type d'utilisation. `result` est une variable de type `Statistiques` et contient donc une référence vers un objet contenant une variable `moyenne` et une variable `variance` ; la notation `result.moyenne` désigne alors la variable `moyenne` de l'objet en question.

9.1.4 Le constructeur par défaut

Intéressons nous maintenant à la ligne 3 de la classe `CalculMoyenneVariance`. Comme nous l'avons vu dans les chapitres précédents, l'utilisation d'objets nécessite le passage par un constructeur. En fait, chaque objet manipulé doit d'abord être créé dans la mémoire de l'ordinateur (dans le tas) avant qu'on puisse obtenir une référence vers celui-ci. Le rôle du **constructeur** est en partie de créer un objet. Nous avons vu dans les chapitres précédents les constructeurs de la classe `StringBuffer` et les constructeurs des tableaux. La ligne 3 de la classe `CalculMoyenneVariance` contient un appel de constructeur complètement similaire à celui utilisé pour les `StringBuffer`s. Son rôle est simplement de créer dans la mémoire de l'ordinateur un objet de type `Statistiques`.

Comme le montre cet exemple, il n'est pas nécessaire de *définir* un constructeur. Si on ne le fait pas, l'ordinateur propose un constructeur sans action réelle, le **constructeur par défaut**. Il ne demande pas de paramètre et ne réalise aucune opération. En effet, le mécanisme de création de l'objet dans la mémoire de l'ordinateur est simplement **déclenché** par l'appel du constructeur. Ce mécanisme réserve la zone mémoire et **initialise toutes les variables de l'objet à zéro**[2]. Ensuite, on exécute le constructeur appelé. Dans le cas du constructeur par défaut, tout ce passe comme si on n'exécutait rien. En dernière analyse, l'appel de la ligne 3 de la classe `CalculMoyenneVariance` a pour simple but de déclencher le mécanisme de création d'un objet instance de `Statistiques`.

[1] nous expliciterons le sens de `public` dans la section 9.3.2

[2] en fait, il s'agit de la valeur correspondant à zéro pour le type de la variable, cf la section 8.5.1

Comme pour les variables locales, il est possible de donner une valeur initiale à une variable d'instance en faisant suivre sa déclaration d'un signe `=` et d'une valeur. On pourrait ainsi modifier la classe `Complexe` de la façon suivante :

Complexe

```
1  public class Complexe {
2    public double re;
3    public double im=1;
4  }
```

Ceci permet d'obtenir un comportement légèrement différent de celui obtenu quand on ne définit de pas de constructeur. Dans la version de base des `Complexe`s, l'appel **`new`** `Complexe()` fabrique le complexe nul, alors que dans la nouvelle version, on obtient i.

9.1.5 Manipulation mémoire et représentation graphique

Considérons le programme suivant :

StatistiquesMemoire

```
1  public class StatistiquesMemoire {
2    public static void main(String[] args) {
3      Statistiques a=new Statistiques();
4      System.out.println(a.moyenne);
5      Statistiques b=new Statistiques();
6      b.variance=0.1;
7      System.out.println(a.variance);
8      System.out.println(b.variance);
9      Statistiques c=a;
10     System.out.println(c.variance);
11     c.moyenne=2;
12     System.out.println(a.moyenne);
13   }
14 }
```

L'affichage produit est le suivant :

Affichage

```
0.0
0.0
0.1
0.0
2.0
```

Analysons les différents lignes obtenues :

- le premier affichage, produit par la ligne 4 du programme, rappelle que, comme nous l'avons dit dans la section précédente, la fabrication d'un objet initialise ses variables à zéro ;
- les deuxième et troisième affichages, produits par les lignes 7 et 8 du programme illustrent un point très important, évoqué à la section 9.1.3 : **chaque** objet possède **sa propre version** des variables d'instance indiquées dans la classe. Ici par exemple, l'objet désigné par `a` et celui désigné par `b` possèdent donc chacun une variable appelée `variance`, ce qui permet de bien comprendre l'affiche produit par le programme ;

– les deux derniers affichages, produits par les lignes 10 et 12 du programme, traduisent un *effet de bord*. Comme nous l'avons longuement étudié dans les chapitres précédents, **tous** les objets sont manipulés par référence. De ce fait, la ligne 9 du programme indique simplement que les variables `a` et `c` ont le même contenu et désignent donc le *même* objet. De ce fait, les notations `a.moyenne` et `c.moyenne` désignent la *même* variable, ce qui permet l'effet de bord observé.

Dans la pratique, les manipulations des objets peuvent rapidement devenir délicates. Pour simplifier l'analyse des programmes, il est utile de recourir, comme dans les chapitres précédents, à une représentation graphique de l'évolution de la mémoire. Pour dessiner un objet dont nous avons écrit la classe, nous placerons à l'intérieur de la case qui le représente, des cases internes représentant chacune des variables d'instance, en utilisant les notations employées pour dessiner la pile. La figure 9.1 donne la représentation de la mémoire après la création du premier objet, c'est-à-dire après l'exécution de la ligne 3 du programme étudié.

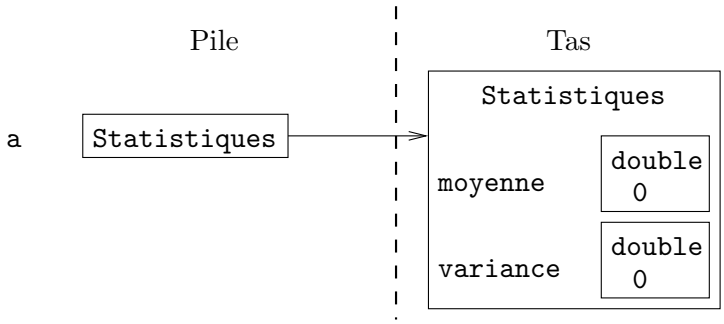

FIG. 9.1 – Un objet `Statistiques` en mémoire

Quand l'ordinateur exécute les lignes suivantes du programme, la mémoire est bien sûr modifiée. Pour bien comprendre l'effet de bord, il suffit de voir que `a` et `c` désignent le même objet. De même, pour bien comprendre le comportement des variables d'instance, il suffit de constater qu'elles sont propres à chaque objet, ce qui est évident sur la représentation graphique (dont l'avantage est justement de souligner le fait que *chaque* objet possède ses propres variables d'instance). La figure 9.2 donne la représentation de la mémoire après l'exécution de la ligne 11 du programme.

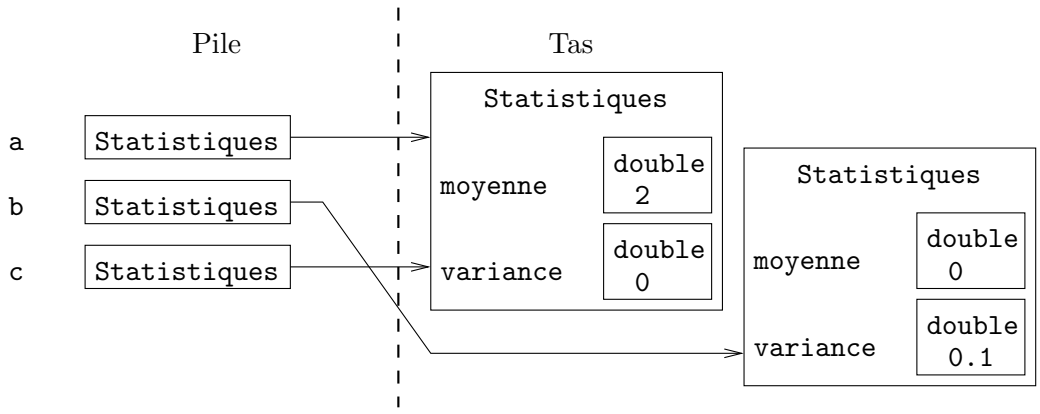

FIG. 9.2 – Deux objets `Statistiques` en mémoire

9.2 Les méthodes d'instance

9.2.1 Motivation

Reprenons l'exemple des nombres complexes, représentés par la classe `Complexe` (voir la section 9.1.3). Il est utile quand on manipule des nombres de complexes de pouvoir calculer le module d'un tel nombre. Pour ce faire, on peut modifier la classe `Complexe` afin d'obtenir la version suivante :

Complexe

```
1 public class Complexe {
2   public double re;
3   public double im;
4   public static double module(Complexe c) {
5     return Math.sqrt(c.re*c.re+c.im*c.im);
6   }
7 }
```

Pour utiliser une telle méthode dans une autre classe, il faut impérativement avoir recours à son nom complet, car c'est une méthode de classe. Voici un exemple d'utilisation :

ComplexeUse

```
1 public class ComplexeUse {
2   public static void main(String[] args) {
3     Complexe c=new Complexe();
4     c.re=1;
5     c.im=0; // inutile, c'est déjà le cas
6     double m=Complexe.module(c);
7     System.out.println(m);
8   }
9 }
```

L'utilisation de la méthode est assez lourde si on la compare à ce qu'il est possible de faire avec les `String`s par exemple. Pour obtenir la longueur d'une chaîne de caractères `s`, il suffit d'écrire `s.length()`, ce qui correspond à l'appel de la méthode *d'instance* `length`. Dans le cas des `Complexe`s, il serait plus agréable de pouvoir écrire `c.module()` pour obtenir le module du complexe `c`. Comme nous l'avons déjà dit au chapitre 7, cette simplification d'écriture est le principal intérêt des méthodes d'instance. Nous allons voir de plus que l'écriture d'une méthode d'instance est en général plus compacte que celle d'une méthode de classe équivalente.

REMARQUE

Il faut se garder de confondre la notion de variable d'instance avec celle de méthode d'instance. En fait, dans les deux notions, on doit entendre instance de façon différente.

Une **variable d'instance** est une variable contenue dans un objet. Chaque objet instance d'une classe donnée possède sa propre version des variables décrite dans la classe. Dans ce contexte, la dénomination "variable d'instance" insiste sur le fait que la variable est *propre* à l'instance.

Une **méthode d'instance** est une méthode qui ne peut être appelée qu'avec un objet appelant. Dans ce contexte, la dénomination "méthode d'instance" ne signifie pas que chaque objet possède sa propre version de la méthode, mais plutôt que la méthode ne peut pas fonctionner sans une instance.

9.2.2 Étude d'un exemple

Considérons une nouvelle version de la classe `Complexe` qui définit maintenant la méthode `module` comme une méthode d'instance :

Complexe

```
1 public class Complexe {
2   public double re;
3   public double im;
4   public double module() {
5     return Math.sqrt(re*re+im*im);
6   }
7 }
```

On remarque les éléments suivants :

- l'en-tête de la méthode ne comporte pas le mot clé `static`. En fait, `static` **indique au compilateur que la méthode déclarée est une méthode de classe**, donc son absence désigne une méthode d'instance ;
- la méthode ne comporte pas de paramètre, ce qui est relativement logique : il s'agit de calculer le module du complexe appelant, il n'y a donc pas lieu d'avoir des paramètres. On retrouver ici le principe de la méthode `length` des `String`s ;
- le texte de la méthode utilise *directement* les variables d'instance, sans préciser l'objet qui les contient.

La dernière remarque est fondamentale : quand on définit une méthode d'instance (c'est-à-dire, techniquement, quand on ne précise pas `static` dans l'en-tête), le compilateur considère qu'on dispose d'un **objet appelant**. Il autorise alors l'**utilisation directe des variables d'instance** : chaque utilisation est interprétée comme une manipulation des variables d'instance de l'objet appelant.

Ce dernier point est délicat à bien saisir, c'est pourquoi nous allons tout de suite présenter la variable spéciale `this` qui éclaire le mécanisme autorisant la définition de méthodes d'instance.

9.2.3 Accès explicite à l'objet appelant par `this`

Il faut garder à l'esprit qu'une méthode d'instance est **toujours** appelée à partir d'un objet, l'*objet appelant*. Dans une méthode d'instance, le compilateur propose une variable particulière appelée `this`. Cette variable est du type de la classe qui définit la méthode et contient toujours une référence vers l'objet appelant. On peut alors réécrire la méthode `module` de la façon suivante :

module

```
1 public double module() {
2   return Math.sqrt(this.re*this.re+this.im*this.im);
3 }
```

Si on admet que l'objet appelant est en fait `this`, cette définition de méthode n'est plus vraiment mystérieuse, et ressemble même énormément à la méthode de *classe* `module` proposée dans la section précédente. Pour mieux comprendre le fonctionnement d'une méthode d'instance, étudions le morceau de programme suivant :

moduleuse

```
1 Complexe c=new Complexe();
2 c.re=1;
3 c.im=0;
4 double m=c.module();
```

Nous avons vu au chapitre 6 le mécanisme de l'appel d'une méthode (de classe). Avant l'exécution du code de la méthode, l'ordinateur recopie les paramètres effectifs dans des variables correspondant aux paramètres formels. Pour les méthodes d'instance, le mécanisme est le même, mais il comporte un paramètre *implicite*, `this`. De ce fait, on peut représenter l'état de la mémoire au début de l'exécution de `module` comme l'indique la figure 9.3.

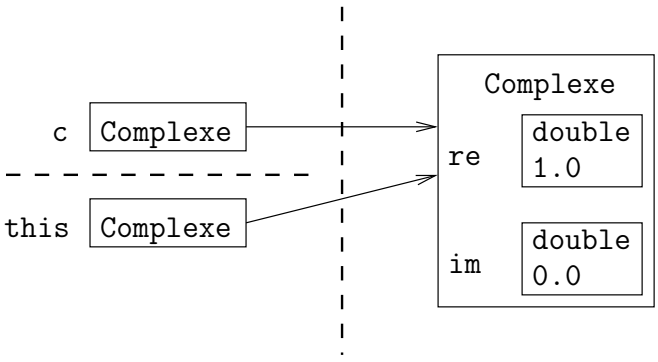

FIG. 9.3 – Appel de la méthode `module`

Le mécanisme qui permet à la méthode d'accéder aux variables de l'objet appelant n'a donc rien de mystérieux. Il se base simplement sur un passage de paramètre *implicite*, sans aucun autre mécanisme. Dans la pratique, comme le paramètre est implicite, on préfère ne pas le faire apparaître, ce qui donne la version de `module` sans `this`, présentée dans la section précédente. Il faut cependant garder à l'esprit qu'il n'y a *aucune différence* entre les deux versions, l'ordinateur se contentant d'ajouter `this` quand il n'est pas présent.

REMARQUE

L'utilisation explicite de `this` n'est pas sans danger. En premier lieu, il est important de s'habituer rapidement à l'écriture sans `this`, car aucun programme réel n'utilise explicitement `this` (sauf dans certains cas particuliers qui dépassent le cadre de ce cours).

En second lieu, `this` n'est pas exactement une variable, mais plutôt une constante. Le programme suivant n'est pas correct et le compilateur ne l'accepte donc pas :

ThisConstante

```
1 public class ThisConstante {
2   int k;
3   public void impossible() {
4     this=new ThisConstante();
5   }
6 }
```

Le compilateur affiche le message d'erreur suivant :

ERREUR DE COMPILATION

```
ThisConstante.java:4: cannot assign a value to final variable this
    this=new ThisConstante();
    ^
1 error
```

Il ne faut cependant pas déduire du caractère constant de `this` celui des variables d'instance. Le programme suivant est parfaitement correct :

ThisConstante2

```
1 public class ThisConstante2 {
2   int k;
3   public void ok() {
4     this.k=2;
5     // équivalent à k=2
6   }
7 }
```

9.2.4 Méthodes à effet de bord

Continuons à étudier la classe `Complexe` en lui ajoutant de nouvelles méthodes. On peut considérer par exemple une méthode qui multiplie le complexe appelant par un réel. Deux solutions sont envisageables. Dans la première, on peut renvoyer un nouvel objet `Complexe` contenant le résultat. Dans la seconde, on peut au contraire *modifier l'objet appelant*. Voici le code de la première solution (qui doit bien entendu être ajouté à la classe `Complexe`) :

Complexe

```
1 public Complexe mult1(double l) {
2   Complexe c=new Complexe();
3   c.re=l*re;
4   c.im=l*im;
5   return c;
6 }
```

Comme toute méthode d'instance, `mult1` possède un paramètre implicite `this`, désignant le `Complexe` appelant. L'ordinateur utilise ce paramètre pour accéder aux variables `re` et `im` qui sont du côté droit des affectations (on pourrait remplacer la ligne 3 par `c.re=l*this.re ;`, ce que fait l'ordinateur). Cette méthode est somme toute relativement classique, si ce n'est le paramètre implicite.

Voici maintenant le code de la seconde solution :

Complexe

```
1 public void mult2(double l) {
2   re*=l;
3   im*=l;
4 }
```

Cette solution est assez différente de la première. Elle se base sur le fait que chaque objet `Complexe` contient deux variables `re` et `im`, dont les contenus seront modifiés par l'appel. Il faut bien comprendre ici la subtilité du problème : dans une méthode classique, la modification d'une variable locale à la méthode n'a en général aucun effet extérieur, car cette variable est indépendante des variables des autres méthodes. Dans une méthode d'instance, la présence du paramètre implicite `this` nous place dans un cas de figure plus complexe, celui des effets de bord. On peut en effet comprendre la ligne 2 de la méthode étudiée comme `this.re*=l ;`. Cette écriture insiste sur le point crucial : la variable qui est modifiée ici n'est pas une variable locale à la méthode. C'est une variable de l'objet appelant et c'est donc le contenu de l'objet qui est modifié par la méthode, exactement comme dans un effet de bord sur les tableaux par exemple. On comprend pourquoi la méthode `mult2` ainsi définie ne renvoie pas de résultat : elle ne doit pas produire de nouveau `Complexe` puisqu'elle *modifie* l'objet appelant.

Pour bien comprendre les différences entre les deux versions de la méthode, étudions un exemple d'utilisation :

ComplexeMultUse

```
1  public class ComplexeMultUse {
2    public static void main(String[] args) {
3      Complexe c=new Complexe();
4      c.re=1;
5      c.im=2;
6      Complexe d=c.mult1(2);
7      System.out.println(d.re);
8      System.out.println(c.re);
9      c.mult2(3);
10     System.out.println(c.re);
11   }
12 }
```

Comme prévu, ce programme affiche :

AFFICHAGE

```
2.0
1.0
3.0
```

Les deux premiers affichages illustrent le fait que la méthode `mult1` produit comme résultat un nouvel objet `Complexe`, indépendant de l'objet appelant, qui n'a pas été modifié lors de l'appel. Au contraire, le dernier affichage montre que `mult2` opère comme convenu en modifiant l'objet appelant.

Au chapitre 7, nous avons vu deux sortes d'objets : les objets immuables (`String`) et les objets modifiables (`StringBuffer`). Nous voyons ici qu'il s'agit avant tout d'une question d'écriture des méthodes, et donc de choix du programmeur. Dans la pratique, on peut vouloir fournir une classe dont les objets ne sont pas modifiables, car on sait que cela permet d'éviter les effets de bord et simplifie donc la programmation (c'est le cas des `String`s). Par contre, on sait que le prix à payer est souvent une perte d'efficacité. Au contraire, les objets modifiables offrent en général de meilleures performances, mais en permettant les effets de bord, ce qui peut rendre leur utilisation plus délicate. Nous continuerons cette discussion dans la suite de ce chapitre (à la section 9.5.4), quand nous pourrons utiliser des techniques permettant de rendre un objet véritable immuable.

9.2.5 Combinaison d'objets

Pour rendre vraiment utile la classe `Complexe` que nous étudions depuis le début de cette section, il faut encore lui ajouter des méthodes. Il est par exemple indispensable de pouvoir faire la somme de deux nombres complexes. Voici une solution envisageable (il faut ajouter cette méthode à la classe `Complexe`) :

somme

```
1  public Complexe somme(Complexe c) {
2    Complexe r=new Complexe();
3    r.re=re+c.re; // équivalent à r.re=this.re+c.re
4    r.im=im+c.im;
5    return r;
6  }
```

Analysons cette méthode :

- le type du résultat est `Complexe` car la méthode produit un nouvel objet `Complexe` ;
- la méthode utilise un seul paramètre de type `Complexe` car c'est une méthode d'instance et elle sera donc appelée par un objet : elle ajoutera donc le complexe représenté par l'objet appelant au complexe représenté par son paramètre. On a donc une dissymétrie entre les paramètres de cette méthode, qu'on retrouve dans son écriture : l'objet appelant est implicite (sauf si on utilise la variable `this`), alors que l'objet paramètre doit apparaître explicitement ;
- le corps de la méthode consiste simplement à calculer les parties réelle et imaginaire du complexe résultat, à créer l'objet correspondant et à le renvoyer.

La solution choisie ne modifie pas l'objet appelant, comme l'illustre le programme de démonstration suivant :

ComplexeSommeUse

```
1  public class ComplexeSommeUse {
2    public static void main(String[] args) {
3      Complexe c=new Complexe(),d=new Complexe();
4      c.re=1;
5      c.im=2;
6      d.re=-3;
7      d.im=1.5;
8      Complexe e=c.somme(d);
9      System.out.println(c.re);
10     System.out.println(c.im);
11     System.out.println(e.re);
12     System.out.println(e.im);
13   }
14 }
```

Comme prévu, ce programme affiche :

AFFICHAGE

```
1.0
2.0
-2.0
3.5
```

L'objet auquel `c` fait référence n'est donc pas modifié, alors que `e` désigne bien la somme des deux complexes `c` et `d`.

Sur le modèle de `somme`, on peut bien entendu construire d'autres méthodes utiles pour la classe `Complexe` : produit, rapport, différence, etc. Ces méthodes ont en commun une caractéristique intéressante : elles combinent des informations provenant de l'objet appelant (`this`) comme toutes les méthodes d'instance, avec des informations provenant d'un ou plusieurs objets paramètres de la méthode.

9.2.6 Constructeurs

Intérêt

Dans toutes les méthodes étudiées dans cette section, nous retrouvons une construction classique :

```
Complexe c=new Complexe();
c.re=2.5;
c.im=-1.5;
```

Si nous comparons cette création d'objet à une construction de tableau ou de `StringBuffer`, elle semble un peu lourde. En effet, pour un tableau, on peut écrire directement :

```
int[] x= new int[] {2,3,4};
```

au lieu de :

```
int[] x= new int[3];
x[0]=2;
x[1]=3;
x[2]=4;
```

En fait, les objets déjà définis en `Java` utilisent tous des *constructeurs*. La définition d'un constructeur permet d'associer à la création d'un objet (qui passe toujours par un `new`) l'initialisation de ses variables d'instance.

Écriture et mécanisme

Voici une nouvelle version de la classe `Complexe`, munie d'un constructeur :

Complexe

```
1 public class Complexe {
2   public double re;
3   public double im;
4   public Complexe(double x,double y) {
5     re=x;
6     im=y;
7   }
8 }
```

La déclaration du constructeur est assez proche de celle d'une méthode, avec les différences suivantes :

- un constructeur porte **obligatoirement le nom de la classe** dans laquelle il est défini ;
- un constructeur **ne doit pas déclarer de type de résultat** ;
- un constructeur **ne peut pas contenir de `return`**.

Ces trois règles doivent impérativement être respectées, car le compilateur refuse un constructeur d'une autre forme. Excepté ces différences finalement assez accessoires, le constructeur se comporte comme une méthode d'instance normale : il a accès aux variables de l'objet (en utilisant éventuellement le paramètre implicite `this`) et se charge en fait de leur donner une valeur initiale (autre que zéro). On peut maintenant remplacer la construction classique d'un `Complexe` par la forme plus compacte suivante :

```
Complexe c=new Complexe(2.5,-1.5);
```

Pour bien comprendre ce qui se passe, il suffit de revenir à l'interprétation du `new`, donnée à la section 9.1.4. Pour exécuter la création qui précède, l'ordinateur effectue les opérations suivantes (les étapes de ce traitement sont illustrées par la figure 9.4) :

1. il crée un objet `Complexe` dans le tas (le type de l'objet créé est fixé par le nom indiqué après le **`new`**) ;

2. il initialise les variables de l'objet à zéro ;
3. il exécute le constructeur choisi :
   (a) il crée ici deux variables locales `x` et `y`, les paramètres formels ;
   (b) il recopie les paramètres effectifs dans les paramètres formels (ici, `2.5` dans `x` et `-1.5` dans `y`) ;
   (c) il prépare le paramètre implicite `this`, qui désigne ici l'objet nouvellement créé (il n'y a pas vraiment d'objet appelant pour un constructeur) ;
   (d) il exécute le code du constructeur (c'est-à-dire copie ici le contenu de `x` dans `re`, et celui de `y` dans `im`, ce qui revient à initialiser la partie réelle du complexe à `2.5` et sa partie imaginaire à `-1.5`) ;
4. il remplace l'appel à `new` par la référence vers l'objet qui vient d'être créé et initialisé.

Constructeur par défaut

Il existe une dernière subtilité dans la définition des constructeurs. Comme nous l'avons vu à la section 9.1.4, l'absence de définition d'un constructeur dans le texte d'une classe entraîne la fabrication automatique par le compilateur d'un constructeur par défaut. Ce constructeur ne fait rien et ne prend donc pas de paramètre. Le point délicat est que **dès qu'on définit un constructeur, le compilateur arrête de fournir le constructeur par défaut**. Dans l'exemple précédent, on a défini un constructeur prenant deux paramètres. De ce fait, le compilateur ne fournit plus de constructeur par défaut sans paramètre. Donc, la ligne :

```
Complexe c=new Complexe();
```

n'est plus valide. Pour obtenir un comportement similaire (création du complexe nul), il faut maintenant écrire :

```
Complexe c=new Complexe(0,0);
```

La conséquence de cette nouvelle contrainte est qu'on doit modifier toutes les méthodes étudiées jusqu'à présent. Par exemple, la méthode `somme` devient :

```
public Complexe somme(Complexe c) {
  return new Complexe(re+c.re,im+c.im);
}
```

On obtient de cette façon sur écriture plus compacte et finalement plus claire. Elle insiste en effet sur la définition mathématique : la somme de deux complexes est un complexe dont la partie réelle (resp. imaginaire) est la somme des parties réelles (resp. imaginaires) des deux complexes de départ.

Plusieurs constructeurs

Une classe peut proposer plusieurs constructeurs adaptés aux données dont le créateur de l'objet dispose au moment du `new`. Chaque constructeur diffère des autres constructeurs par ses paramètres (par leurs types et/ou par leur nombre). Dans le cas des complexes par exemple, nous pourrions ajouter un constructeur sans paramètre, permettant de fabriquer le complexe nul. Il faudrait ajouter le texte suivant à la classe `Complexe` :

Complexe

```
1 public Complexe() {
2   // il n'y a rien à faire, les variables contiennent déjà zéro
3 }
```

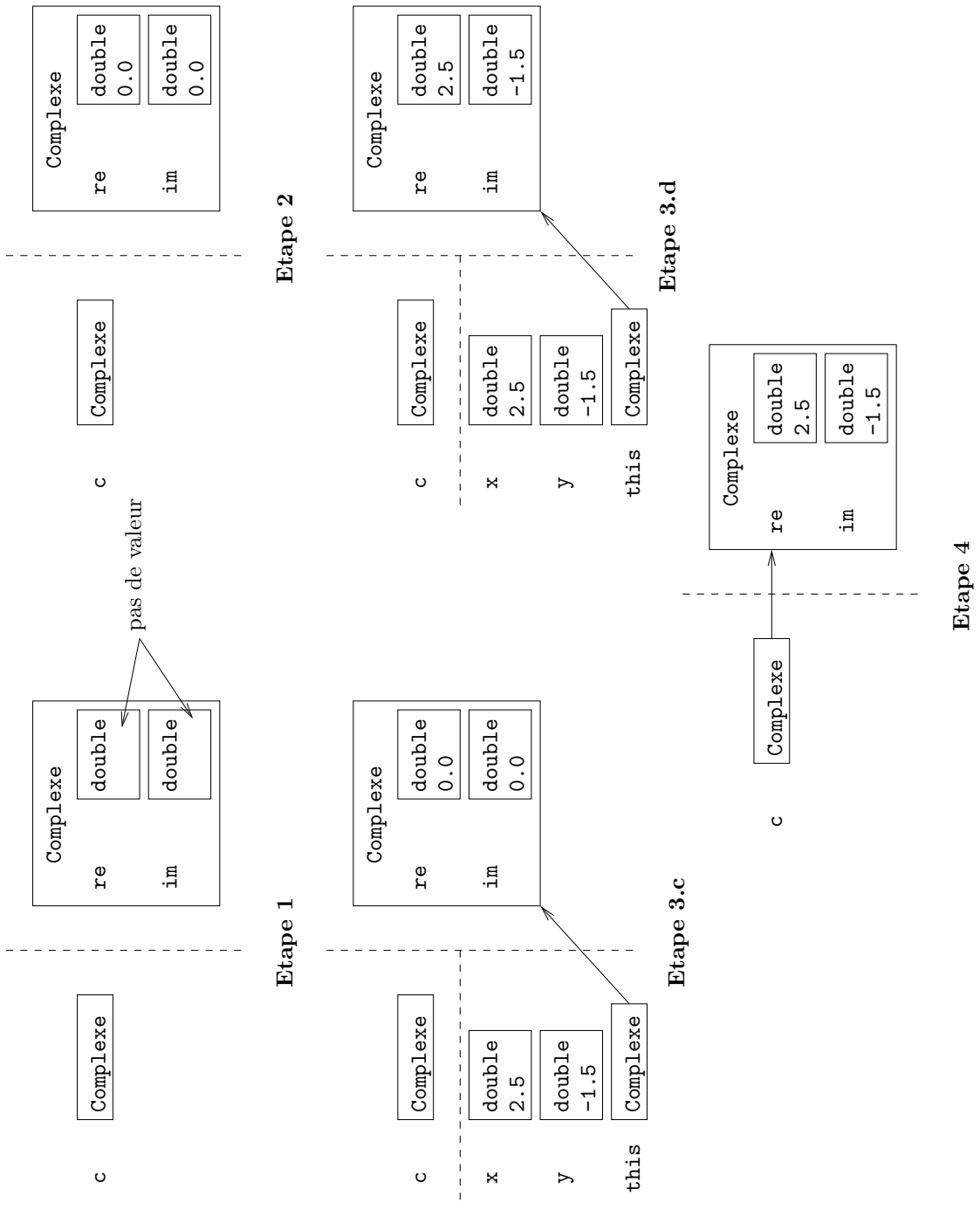

FIG. 9.4 – Les étapes de l'exécution d'un constructeur

9.2.7 Représentation textuelle d'un objet

Dans les exemples qui précédent, nous avons toujours utilisé directement les variables d'instance `re` et `im` afin d'afficher les composantes d'un objet représentant un nombre complexe. On peut se demander ce que provoque l'utilisation d'une commande du type `System.out.println(a) ;` où `a` désigne une variable de type `Complexe`. En fait, l'ordinateur va afficher une ligne de la forme :

```
Complexe@8162588
```

Le nombre situé après `Complexe` dépend du programme[3], mais la forme générale, `Complexe@` suivi d'un nombre entier est toujours la même et ne procure pas une solution satisfaisante pour l'affichage d'un nombre complexe. De façon générale, quand on définit une classe `A`, l'affichage d'un objet de type `A`, se traduit par `A@` suivi d'un nombre entier.

Fort heureusement, il existe un moyen de remédier à cet inconvénient. Il consiste à écrire une méthode `toString()` renvoyant une `String` censée représenter l'objet appelant. Ajoutons par exemple la méthode suivante :

toString

```
1 public String toString() {
2   return "("+re+","+im+")";
3 }
```

On peut ainsi "convertir" un objet `Complexe` en une chaîne de caractères qui représente ses parties entière et imaginaire sous forme d'un couple. Étudions le comportement du programme suivant :

TestComplexeToString

```
1 public class TestComplexeToString {
2   public static void main(String[] args) {
3     Complexe a=new Complexe(1,2);
4     System.out.println(a.toString());
5     System.out.println(a);
6   }
7 }
```

L'affichage obtenu est :

Affichage

```
(1.0,2.0)
(1.0,2.0)
```

En fait, quand on cherche à afficher un objet quelconque (grâce à la méthode `println` par exemple), l'ordinateur détermine si celui-ci propose une méthode `toString`. Si c'est le cas, cette méthode est utilisée pour obtenir une forme affichable pour l'objet. Dans le cas contraire, une méthode par défaut est utilisée (c'est elle qui donne l'affichage obtenu au départ). Dans tous les cas, il y a donc un appel "automatique[4]" à *une* méthode `toString` de conversion.

Il faut être extrêmement attentif à un point : il faut **impérativement** définir une méthode nommée `toString` (en respectant scrupuleusement l'orthographe). Cette méthode doit **impérativement** renvoyer une `String` et ne pas prendre de paramètre. Pour voir ce qui se passe quand on ne respecte pas ces consignes, reportez-vous à l'exemple 9.1.

[3] En fait, il s'agit d'un entier fabriqué par le compilateur à partir de l'objet. Si on affiche plusieurs fois le même objet, on obtient à chaque fois le même entier.

[4] en réalité, l'appel n'a rien d'automatique. Cependant, le mécanisme utilisé se base sur la notion d'*héritage* qui dépasse le cadre de ce cours.

REMARQUE

Il est important de bien comprendre que l'appel à `toString` ne réalise **aucun affichage** en lui-même. C'est son utilisation automatique par l'ordinateur qui permet l'affichage. Plus précisément, l'ordinateur est capable d'afficher les `String`s. Si on demande l'affichage d'un objet quelconque, l'ordinateur va commencer par obtenir une représentation de cet objet sous forme d'une `String` (grâce à la méthode `toString`) puis va effectivement afficher cette représentation.

Il faut noter de plus qu'il est absurde de faire un affichage dans la méthode `toString`, car cela reviendra dans la pratique à faire deux affichages (cf exemple 9.1). Il est d'autre part inutile d'appeler explicitement la méthode `toString`, le compilateur s'en chargeant automatiquement.

Voici des exemples de **mauvaises** définitions de `toString` (**à ne pas reproduire**, bien entendu) :

Exemple 9.1 :

Toute la confusion provient en général du fait que `toString` est utilisé pour l'affichage, alors qu'en fait cette méthode se contente de transformer l'objet appelant en une chaîne de caractères. Voici donc un objet qui définit une méthode `toString` d'affichage :

BadToString

```
1  public class BadToString {
2    private int x;
3    public BadToString(int y) {
4      x=y;
5    }
6    public String toString() {
7      System.out.println(x);
8      return "("+x+")";
9    }
10   public static void main(String[] args) {
11     BadToString b=new BadToString(2);
12     System.out.println(b);
13   }
14 }
```

Ce programme affiche deux fois le contenu de l'objet :

AFFICHAGE

```
2
(2)
```

Le premier affichage est celui déclenché par la ligne 7, alors que le second est produit par la ligne 12. On peut faire pire que cette première erreur, par exemple en ne déclarant pas `toString` comme on doit le faire :

BadToString2

```
1  public class BadToString2 {
2    private int x;
3    public BadToString2(int y) {
4      x=y;
5    }
6    public void toString() {
7      System.out.println(x);
```

```
8   }
9   public static void main(String[] args) {
10    BadToString2 b=new BadToString2(2);
11    System.out.println(b);
12    System.out.println(b.toString());
13  }
14 }
```

Ce programme ne compile pas, car il comporte deux erreurs. Le compilateur affiche le message suivant :

Erreur de compilation

```
BadToString2.java:6: toString() in BadToString2 cannot override toString()
in java.lang.Object; attempting to use incompatible return type
found   : void
required: java.lang.String
  public void toString() {
              ^
BadToString2.java:12: 'void' type not allowed here
    System.out.println(b.toString());
                                   ^
2 errors
```

La ligne 12 n'est pas correcte car la méthode `toString` qui est définie ne renvoie pas de résultat (`void`) et on ne peut donc pas afficher le "résultat" d'un appel à cette méthode. L'autre erreur est plus complexe, et le compilateur affiche à son propos un message peu clair. Pour simplifier, ce message indique qu'on ne définit par correctement `toString` qui doit renvoyer une `String`. Malheureusement, on peut faire d'autres erreurs que le compilateur ne détecte pas :

BadToString3

```
1  public class BadToString3 {
2    private int x;
3    public BadToString3(int y) {
4      x=y;
5    }
6    public String tostring() {
7      return String.valueOf(x);
8    }
9    public static void main(String[] args) {
10     BadToString3 b=new BadToString3(2);
11     System.out.println(b);
12   }
13 }
```

Ce programme affiche :

Affichage

```
BadToString3@23bcc1
```

En effet, comme la méthode définie s'appelle `tostring`, sans majuscule à `String`, le compilateur ne la reconnaît pas. Il utilise donc la version de base, ce qui donne l'affichage indiqué.

La méthode `toString` est utilisée dès qu'on doit obtenir une représentation par une chaîne de caractères d'un objet. C'est en particulier le cas quand on utilise l'opérateur `+` avec des chaînes de caractères, comme le montre le programme suivant :

TestComplexeToString2

```
1  public class TestComplexeToString2 {
2    public static void main(String[] args) {
3      Complexe a=new Complexe(1,2);
4      String s="->"+a+"<-";
5      StringBuffer sb=new StringBuffer("<");
6      sb.append(a);
7      sb.append(">");
8      String t=String.valueOf(a);
9      System.out.println(s);
10     System.out.println(sb);
11     System.out.println(t);
12   }
13 }
```

Le programme affiche :

Affichage

```
->(1.0,2.0)<-
<(1.0,2.0)>
(1.0,2.0)
```

En fait, la méthode `toString` est appelée automatiquement dans les conditions suivantes :
- quand on transmet l'objet aux méthodes `print` et `println` de `System.out` ;
- quand on transmet l'objet à la méthode d'instance `append` des `StringBuffers` ;
- quand on transmet l'objet à la méthode de classe `valueOf` des `Strings` ;
- quand on cherche à ajouter l'objet à un `String` avec l'opérateur `+` de concaténation.

Il faut noter pour finir qu'il est impossible de placer directement un objet dans une variable de type `String` ou encore de transtyper un objet en `String`. Les deux dernières lignes du texte suivant sont donc rejetées par le compilateur :

```
Complexe c=new Complexe(1,2);
String s=c;
String t=(String)c;
```

9.2.8 Comparaison : la méthode `equals`

Nous avons vu à la section 7.4.8 que l'opérateur `==` comparait les contenus des variables, c'est-à-dire les références, et non pas les objets. Considérons les lignes suivantes :

```
Complexe c=new Complexe(1,2);
Complexe d=new Complexe(1,2);
Complexe e=c;
System.out.println(c==d);
System.out.println(c==e);
```

L'affiche produit est :

Affichage

```
false
true
```

En effet, les deux objets auxquels `c` et `d` font référence sont distincts et les contenus des variables (les références) sont dont différents, ce qui donne le premier affichage. Pour le second, il s'agit d'une évidence : les variables `e` et `c` contiennent la même valeur à cause de la ligne 3 du texte.

Nous avons vu dans la section 7.3.7 que les `String`s peuvent être comparées par l'intermédiaire d'une méthode `equals` qui les comparent "vraiment", c'est-à-dire en se basant sur le contenu des objets, et pas sur les références. On peut être tenté d'essayer si une telle méthode existe pour les `Complexe`s :

```
Complexe c=new Complexe(1,2);
Complexe d=new Complexe(1,2);
Complexe e=c;
System.out.println(c.equals(d));
System.out.println(c.equals(e));
```

Deux choses étonnantes se produisent : ce programme compile et il affiche les lignes suivantes :

Affichage

```
false
true
```

On peut donc en déduire que **toute classe propose une méthode `equals`** et qu'en général, cette méthode **est équivalente à une comparaison des références**. En fait, nous sommes ici dans une situation complètement similaire à celle de la méthode `toString` : si on ne définit pas notre propre méthode `equals`, le compilateur fournit automatiquement une méthode qui se base sur la comparaison des références. Il est cependant important de comprendre **qu'il n'y a pas d'appel automatique à `equals`** : le compilateur ne remplace pas `==` par un appel à `equals`. Plus simplement, si on utilise une telle méthode et qu'on ne l'a pas définie, alors le compilateur fournit une version simple.

On peut se demander comment programmer une méthode `equals` aussi utile que celle des `String`s. Malheureusement, c'est assez délicat, comme le montre l'exemple de la méthode suivante, qui doit être ajoutée à la classe `Complexe` :

equals

```
1 public boolean equals(Object c) {
2   if(c instanceof Complexe) {
3     Complexe a=(Complexe)c;
4     return a.re==re && a.im==im;
5   } else {
6     return false;
7   }
8 }
```

La ligne 4 est claire, il s'agit de la véritable comparaison entre l'objet appelant et l'objet paramètre. Le reste est assez obscur et ne peut pas être expliqué sans sortir du cadre du présent ouvrage. Cette forme générale est cependant **indispensable**. Il est en particulier obligatoire d'utiliser la première ligne sans rien y changer (excepté éventuellement le nom du paramètre formel). On doit effectuer le

test de la ligne 2 et le transtypage de la ligne 3, en l'adaptant à la classe dans laquelle on souhaite programmer une méthode `equals`. On peut admettre que les trois premières lignes s'arrangent pour proposer dans la variable `a` une référence vers l'objet paramètre de la méthode. La ligne 4 ne pose alors pas de problème.

9.3 Le contrôle d'accès

9.3.1 Motivations

Comme son nom l'indique, le **contrôle d'accès** est un dispositif permettant au programmeur de contrôler comment les différentes méthodes qui constituent un programme `Java` peuvent manipuler les variables et/ou les méthodes définies par certaines classes.

Nous avons vu au chapitre 7 que l'utilisation d'objets *immuables* est plus facile que celle d'objets modifiables (cf. en particulier la section 7.4.6). Nous venons de voir dans la section précédente qu'il est souvent possible de programmer les méthodes d'instance d'une classe de sorte qu'elles ne modifient pas l'objet appelant, conservant pour celui-ci un caractère immuable (cf. la section 9.2.4). Pourtant, nous ne pouvons toujours pas obtenir des objets non modifiables, comme le montre l'exemple suivant :

Exemple 9.2 :

Reprenons comme support les `Complexe`s :

ComplexeModifiable

```
1 public class ComplexeModifiable {
2   public static void main(String[] args) {
3     Complexe c=new Complexe(1,2),d=c;
4     System.out.println(c);
5     d.re=3;
6     System.out.println(c);
7   }
8 }
```

Comme prévu, ce programme affiche :

AFFICHAGE

```
(1.0,2.0)
(3.0,2.0)
```

Il y a bien eu un effet de bord, du à la modification de l'objet auquel `c` et `d` font référence (ligne 5 du programme).

Une motivation importante (à notre niveau) du contrôle d'accès est de supprimer ce problème en permettant d'interdire la modification directe d'un objet. Une autre motivation apparaîtra dans les exemples traités dans cette section : la modification d'un objet peut non seulement le rendre plus délicat à utiliser à cause d'éventuels effets de bord, mais elle peut aussi le rendre inutilisable (cf 9.3.4), ce qui est nettement plus gênant. Nous aborderons des justifications plus générale du contrôle d'accès dans la section 9.4 sur l'intérêt des objets.

9.3.2 Le mot clé `private`

Le contrôle d'accès se met en place grâce à l'utilisation d'un nouveau mot clé : **`private`**. A chaque fois que nous avons utilisé le mot clé **`public`**, nous aurions pu utiliser à la place `private`.

Pour bien comprendre le sens de ces mots clé, il faut revenir brièvement sur la notion d'accès à une variable et d'appel de méthode.

Une classe comporte la définition d'un certain nombre de ressources : des méthodes de classe, des variables et des méthodes d'instance. L'accès à chaque ressource est contrôlé par le compilateur et est basé sur un principe simple : la classe qui définit une ressource indique par un mot clé le mode d'accès qu'elle choisit pour cette ressource. Une ressource `public` est accessible sans contrôle. Cela signifie qu'il est toujours possible d'appeler une méthode `public` ou d'accéder à une variable d'instance `public`. Au contraire, **une ressource `private` est d'accès interdit en dehors de la classe qui la définit**. En d'autres termes, si une variable d'instance définie par la classe `A` est `private`, il est impossible de la manipuler dans une méthode d'une classe autre que `A`.

Donnons un exemple simple pour bien comprendre la situation :

Exemple 9.3 :

Commençons par définir une classe dans laquelle on propose une variable d'instance **`public`** et une autre **`private`** :

PublicPrivate

```
1  public class PublicPrivate {
2    public int a;
3    private int b;
4    public PublicPrivate(int u,int v) {
5      a=u;
6      b=v;
7    }
8    public String toString() {
9      return "("+a+","+b+")";
10   }
11 }
```

On remarque que la méthode `toString` utilise la variable d'instance `b`. Ceci est parfaitement possible car la méthode `toString` est définie par la *même* classe que la variable en question. Or, le mot clé **`private`** interdit l'accès *en dehors de la classe de définition de la ressource.*

Considérons maintenant une tentative d'utilisation illicite :

UsePublicPrivate

```
1  public class UsePublicPrivate {
2    public static void main(String[] args) {
3      PublicPrivate pp=new PublicPrivate(3,5);
4      System.out.println(pp);
5      pp.a=7;
6      System.out.println(pp);
7      pp.b=5;
8    }
9  }
```

Quand on tente de compiler ce programme, l'ordinateur refuse et affiche le message d'erreur suivant :

Erreur de compilation

```
UsePublicPrivate.java:7: b has private access in PublicPrivate
    pp.b=5;
      ^
1 error
```

Nous sommes en effet dans une classe différente de la classe de définition de la ressource concernée (la variable d'instance `b`) et l'accès est donc interdit. Par contre, le reste du programme est parfaitement correct, car il n'utilise que des ressources **public** : le constructeur, la méthode `toString` et la variable d'instance `a`. Le programme sans la dernière ligne affiche d'ailleurs :

AFFICHAGE

```
(3,5)
(7,5)
```

Il faut bien comprendre que **l'accès est contrôlé au niveau de la classe et pas au niveau de l'instance**, comme le montre l'exemple suivant :

Exemple 9.4 :

AccesClasse

```
1  public class AccesClasse {
2    private double x;
3    public AccesClasse(double u) {
4      x=u;
5    }
6    public AccesClasse somme(AccesClasse a) {
7      return new AccesClasse(x+a.x);
8    }
9    public String toString() {
10     return String.valueOf(x);
11   }
12   public static void main(String[] args) {
13     AccesClasse a=new AccesClasse(3);
14     System.out.println(a);
15     AccesClasse b=a.somme(a);
16     System.out.println(b);
17     b.x=3.5;
18     System.out.println(b);
19   }
20 }
```

Ce programme compile parfaitement. En effet, la ligne 7 ne pose pas de problème, car on accède à la variable `x` d'un objet instance de `AccesClasse` dans une méthode définie dans la même classe. Le fait que `a.x` désigne une variable de l'objet paramètre (et pas de l'objet appelant) ne pose aucun problème. De même, la ligne 17 ne pose aucun problème, car la méthode `main` est définie dans la même classe que la variable d'instance `x` de l'objet auquel `b` fait référence. La méthode `main` affiche :

AFFICHAGE

```
3.0
6.0
3.5
```

On remarque que la classe `AccesClasse` comporte à la fois des méthodes d'instance et des méthodes de classe, ce qui est parfaitement correct et parfois très pratique, comme nous le verrons dans la suite du chapitre.

9.3.3 Mise en oeuvre pratique : constructeurs et méthodes d'accès

En déclarant **private** toutes les variables d'instance d'une classe, il devient impossible de modifier un objet de cette classe en dehors de celle-ci. On peut ainsi obtenir des objets immuables, comme les `String`s, à condition bien sûr de proposer exclusivement des méthodes d'instance qui ne modifient pas leur objet appelant. Dans la pratique, il reste cependant deux petits problèmes :

1. il faut impérativement proposer un constructeur. En effet, si aucune méthode ne permet de modifier les valeurs des variables d'instance, celles ci contiennent les valeurs données au moment de la construction de l'objet. Si on ne fournit pas de constructeur, les variables d'instance contiennent alors toutes zéro, ce qui n'est pas très utile en général ;
2. il est parfois utile de fournir des **observateurs**. Un observateur est une méthode qui permet d'obtenir la valeur d'une variable d'instance de l'objet appelant, ou plus généralement d'obtenir une information sur l'objet appelant, sans pour autant pouvoir le modifier. Cette possibilité est parfois cruciale. Si on considère l'exemple de la classe `AccesClasse`, on remarque qu'il est impossible d'écrire `a.x` dans une classe autre que `AccesClasse`, si `a` est une référence vers un objet de type `AccesClasse`. Cela signifie bien sûr qu'on ne peut pas modifier l'objet, mais cela signifie aussi qu'il n'est pas possible d'obtenir le contenu de la variable `x`, même pour un affichage anodin, par exemple.

Exemple 9.5 :

Voici comment on peut modifier la classe `Complexe` déjà longuement étudiée pour la rendre *immuable* :

Complexe

```
1  public class Complexe {
2    private double re;
3    private double im;
4    public Complexe(double x,double y) {
5      re=x;
6      im=y;
7    }
8    public double getRéelle() {
9      return re;
10   }
11   public double getImaginaire() {
12     return im;
13   }
14   public double module() {
15     return Math.sqrt(re*re+im*im);
16   }
17   public Complexe somme(Complexe c) {
18     return new Complexe(re+c.re,im+c.im);
19   }
20   public String toString() {
21     return "("+re+","+im+")";
22   }
23   public boolean equals(Object c) {
24     if(c instanceof Complexe) {
25       Complexe a=(Complexe)c;
26       return a.re==re && a.im==im;
27     } else {
```

```
28          return false;
29        }
30      }
31    }
```

Les deux variables d'instance sont maintenant déclarées **private**, ce qui interdit leur manipulation en dehors de la classe `Complexe`. On voit qu'excepté le constructeur, aucune méthode ne modifie les variables d'instance. De ce fait, un objet `Complexe` est complètement immuable et aucun effet de bord n'est possible.

Pour permettre une utilisation pratique des `Complexe`s, on a ajouté deux observateurs, les méthodes `getRéelle` et `getImaginaire`[5], qui permettent respectivement d'obtenir la partie réelle et la partie imaginaire du `Complexe` appelant. Ces méthodes sont indispensables si on veut pouvoir obtenir les informations en question, étant donné qu'il est impossible d'écrire `c.re` dans une classe autre que `Complexe`, si `c` contient une référence vers un objet `Complexe`.

REMARQUE

La présence des observateurs est parfois source d'erreur. Considérons par exemple la méthode d'instance `getImaginaire` des `Complexe`s. Cette méthode renvoie la partie imaginaire du complexe appelant. Comme le type de son résultat est un type fondamental (en l'occurrence `double`), la méthode renvoie une **valeur**. De ce fait, écrire une instruction de la forme `a.getImaginaire()=1.5 ;` (où `a` désigne une variable de type `Complexe`) n'a absolument **aucun sens**. En fait, si par exemple la partie imaginaire du complexe considéré vaut `2.3`, ceci revient à écrire `2.3=1.5 ;` !

On pourrait croire que comme `getImaginaire` est définie grâce au contenu d'une variable, l'ordinateur se "souvient" d'une manière ou d'une autre de cette définition et peut donc "revenir en arrière". C'est totalement faux et profondément absurde.

9.3.4 Exemple d'application : les rationnels

Pour représenter informatiquement un nombre rationnel, il suffit de deux entiers. On pourrait donc se contenter d'un tableau de type `int[]`, ce qui poserait bien sûr le problème de l'ordre de stockage. Pour rendre celui-ci explicite, on passe par un objet, avec deux variables d'instance, `d` le dénominateur et `n` le numérateur. Le seul problème est qu'il est impératif d'avoir un dénominateur non nul. De plus, il est très utile de représenter les rationnels sous forme réduite, c'est-à-dire telle que le numérateur et le dénominateur soient premiers entre eux. Pour ces raisons, il est indispensable d'utiliser le contrôle d'accès : il va nous permettre d'empêcher une modification intempestive du dénominateur et donc éviter les rationnels non définis. De plus, il va nous permettre de ne manipuler que des rationnels réduits (toujours en empêchant les modifications).

Il nous faut dans un premier temps régler le problème de la réduction d'un rationnel. Étant donnés p et q deux entiers, l'écriture irréductible du rationnel $\frac{p}{q}$ s'obtient en divisant les deux entiers par leur Plus Grand Commun Diviseur (PGCD). Pour calculer le PGCD de deux entiers, on applique l'algorithme d'Euclide, basé sur la propriété suivante : si $r > 0$ est le reste de la division de p par q, on a $\mathrm{PGCD}(p,q) = \mathrm{PGCD}(q,r)$. Bien entendu, si le reste est nul, c'est que q divise p et donc que $\mathrm{PGCD}(p,q) = q$. Pour calculer le PGCD, il suffit donc d'appliquer l'identité de façon itérée, car la décroissance stricte du deuxième terme du couple étudié (par définition, le reste r est strictement plus petit que q) implique qu'au bout d'un nombre fini d'application de l'identité, on va atteindre le cas $r = 0$. On peut donc écrire la méthode suivante :

[5] la tradition en `Java` est de donner un nom qui commence par `get` aux observateurs.

pgcd

```
1  public static int pgcd(int p,int q) {
2    int r;
3    do {
4      r=p%q;
5      p=q;
6      q=r;
7    } while(r>0);
8    return p;
9  }
```

On remarque que si `q` est nul, cette méthode va "planter", car on va essayer de faire une division par zéro. Ce comportement est exactement celui souhaité pour la classe `Rationnel`, comme nous allons le voir par la suite.

Pour la classe `Rationnel` en elle-même, il nous faut simplement prévoir deux variables d'instance, un constructeur, des observateurs pour les variables d'instance, et des méthodes permettant de rendre la classe vraiment utilisable :

- il important de prévoir une méthode `toString` ;
- on doit prévoir un moins une méthode `somme` et une méthode `produit` ;
- il est utile de prévoir une méthode `toDouble` qui propose la valeur numérique approchée du rationnel ;
- il est utile de programmer une méthode `equals` pour permettre la comparaison de deux rationnels.

Bien entendu, une classe complète demanderait d'autres méthodes, comme par exemple des méthodes d'inversion, de multiplication par un entier, etc. Voici le texte proposé pour la classe :

Rationnel

```
1  public class Rationnel {
2    private int n; // numérateur
3    private int d; // dénominateur
4    public Rationnel(int p,int q) {
5      int div=pgcd(p,q);
6      n=p/div;
7      d=q/div;
8      if(d<0) {
9        d=-d;
10       n=-n;
11     }
12   }
13   public int getNumérateur() {
14     return n;
15   }
16   public int getDénominateur() {
17     return d;
18   }
19   public double toDouble() {
20     return ((double)n)/d;
21   }
22   public Rationnel somme(Rationnel r) {
23     return new Rationnel(n*r.d+r.n*d,r.d*d);
```

```
24    }
25    public Rationnel produit(Rationnel r) {
26      return new Rationnel(n*r.n,r.d*d);
27    }
28    public String toString() {
29      if(d==1)
30        return String.valueOf(n);
31      return n+"/"+d;
32    }
33    public boolean equals(Object o) {
34      if(o instanceof Rationnel) {
35        Rationnel r=(Rationnel)o;
36        return r.n==n && r.d==d;
37      } else {
38        return false;
39      }
40    }
41    public static int pgcd(int p,int q) {
42      int r;
43      do {
44        r=p%q;
45        p=q;
46        q=r;
47      } while(r>0);
48      return p;
49    }
50    public static void main(String[] arg) {
51      Rationnel r=new Rationnel(1,2);
52      System.out.println(r);
53      System.out.println(r.toDouble());
54      Rationnel p=r.produit(r);
55      System.out.println(p);
56      Rationnel s=r.somme(r);
57      System.out.println(s);
58      System.out.println(new Rationnel(4,-2));
59      r=p.produit(r).somme(r);
60      System.out.println(r);
61      Rationnel t=new Rationnel(5,8);
62      System.out.println(t==r);
63      System.out.println(t.equals(r));
64    }
65  }
```

Il n'y a rien de bien particulier dans cette classe, excepté peut être la présence de la méthode `pgcd` et l'écriture du constructeur :

- la méthode de classe `pgcd` pourrait bien entendu être définie dans une autre classe. Cependant, elle est particulièrement utile dans le cadre du traitement des rationnels, et il est donc relativement logique de la placer dans la classe `Rationnel`. Si on l'utilise dans une méthode d'une autre classe, on devra bien sûr passer par son nom complet, `Rationnel.pgcd` ;

- le constructeur effectue une mise sous forme irréductible (en exigeant de plus un dénominateur strictement positif), ce qui peut sembler étrange. En fait, c'est extrêmement pratique : grâce à cette écriture, il est impossible d'avoir des rationnels non réduits, et on n'a plus besoin de se poser la question de la réduction dans les méthodes `somme` et `produit`. De plus, l'appel à `pgcd` provoque une erreur si on tente de créer un rationnel avec un dénominateur nul, ce qui est parfait. Notons enfin que cela permet, dans la méthode `toString`, de faire une conversion qui supprime le dénominateur quand il vaut 1 ;
- notons pour finir la présence de la méthode `main`, qui permet de tester la classe et qui donne l'affichage suivant :

AFFICHAGE

```
1/2
0.5
1/4
1
-2
5/8
false
true
```

Sans contrôle d'accès, toute la construction mise en place est caduque, en particulier le traitement effectué dans le constructeur. En effet, l'utilisateur peut alors modifier directement les variables d'instance et donc obtenir un rationnel non réduit, ou pire encore non définit (en donnant une valeur nulle au dénominateur).

9.4 Intérêt des objets

La *programmation objet*, c'est-à-dire le fait de concevoir et développer un programme comme un ensemble d'objets en interaction, a de nombreux avantages par rapport aux anciennes formes de programmation (notamment la programmation dite *procédurale*). Nous ne pourrons pas tous les illustrer à ce niveau du cours, car certains avantages sont liés à des techniques de programmation évoluées que nous n'avons pas encore abordées. Il est cependant possible d'évoquer brièvement certains avantages.

Il est important de noter qu'il existe une justification plus pragmatique de l'écriture d'objets en `Java` : certaines opérations sont impossibles à réaliser autrement dans ce langage ! C'est le cas par exemple du graphisme, que nous étudierons chapitre 10.

9.4.1 Créer des abstractions

Grâce aux objets, il est possible de représenter informatiquement des entités du monde réel, en particulier (mais pas exclusivement) des concepts mathématiques importants. Sans les objets, on serait par exemple obligé de représenter les vecteurs de $\mathbb{R}^n$ et les polynômes de degré inférieur ou égal à $n-1$ de la même façon, c'est-à-dire comme des tableaux de réels. La conséquence directe de cette confusion serait la possibilité d'interpréter de façon incorrecte un tableau dans une partie du programme (confondre un vecteur avec un polynôme par exemple). De plus, si la numérotation des coefficients d'un polynôme à partir de 0 est parfaitement justifiée mathématiquement, il n'en va pas de même pour les coordonnées d'un vecteur.

De ce fait, même pour des concepts mathématiques simples, la notion d'objet est indispensable car elle empêche la confusion des objets entre eux et permet de fournir pour chaque objet un

ensemble de méthodes adaptées à son utilisation. Pour un objet `Vecteur`, on proposera par exemple une méthode `coor` qui à un paramètre entier n associe la valeur de la n-ième coordonnée du vecteur correspondant, en numérotant les coordonnées à partir de 1. Pour un objet `Polynome`, on proposera une méthode `coeff` qui à un paramètre entier n associe le coefficient du terme X^n dans le polynôme correspondant (avec bien sûr n commençant ici à 0). De ce fait, les méthodes d'un objet permettent d'avoir un ensemble d'outils naturels pour la manipulation de l'entité représentée.

Quand on souhaite représenter des entités de la vie courante, le problème se pose avec encore plus d'acuité. En effet, on est parfois amené à regrouper des valeurs de natures diverses pour décrire une entité particulière. Si on souhaite par exemple représenter informatiquement un étudiant pour gérer son dossier, il est naturel de conserver son nom (une chaîne de caractères), son prénom (une chaîne de caractères), sa date de naissance (trois entiers), etc. Or, il n'existe aucun moyen autre que les objets de regrouper entre elles des variables de natures différentes. De même, seule l'utilisation d'un objet permet de bien interpréter les trois entiers qui forment la date de naissance, grâce à une méthode adaptée.

Pour finir, un intérêt crucial des objets est de masquer la façon dont une entité est représentée. Il est ainsi parfaitement possible de modifier la classe `Complexe` afin de stocker les parties réelle et imaginaire dans un tableau de réels. On doit bien sûr reprogrammer les méthodes, mais les programmes qui utilisent la classe `Complexe` restent exactement les mêmes. De ce fait, on crée bien une abstraction informatique pour représenter une entité du monde réel, car l'utilisateur de la représentation informatique n'a pas besoin de savoir concrètement comment elle est programmée : c'est le cas par exemple de la classe `String` incluse dans `Java` sans qu'on sache vraiment comment elle est programmée.

Exemple 9.6 :

Voici par exemple une nouvelle programmation (simplifiée) de la classe `Complexe`. Les parties réelle et imaginaire sont stockées dans un tableau. Les méthodes sont donc programmées de façon assez différente de celle utilisée pour la classe `Complexe` d'origine. Par contre, d'un point de vue extérieur, la nouvelle classe est identique à l'ancienne (mêmes méthodes, même constructeur) :

Complexe

```
1  public class Complexe {
2    private double[] coor;
3    public Complexe(double x,double y) {
4      coor=new double[] {x,y};
5    }
6    public double getRéelle() {
7      return coor[0];
8    }
9    public double getImaginaire() {
10     return coor[1];
11   }
12   public double module() {
13     return Math.sqrt(coor[0]*coor[0]+coor[1]*coor[1]);
14   }
15   public Complexe somme(Complexe b) {
16     return new Complexe(coor[0]+b.coor[0],coor[1]+b.coor[1]);
17   }
18   public String toString() {
19     return "("+coor[0]+","+coor[1]+")";
20   }
```

```
21    public boolean equals(Object c) {
22      if(c instanceof Complexe) {
23        Complexe a=(Complexe)c;
24        return a.coor[0]==coor[0] && a.coor[1]==coor[1];
25      } else {
26        return false;
27      }
28    }
29  }
```

9.4.2 Séparer les problèmes

Comme nous venons de l'évoquer, le recours aux objets permet de masquer la façon dont une entité du monde réel est représentée informatiquement. En fait, une classe décrit une **interface**, c'est-à-dire un ensemble de méthodes (constructeurs inclus) permettant de créer et d'utiliser des objets. Pour faire fonctionner un programme en faisant interagir des objets, il suffit de savoir quel sens accorder à chaque méthode. Il importe peu de savoir comment chaque objet est programmé. Ceci permet de découper la résolution d'un problème en plusieurs parties, chacune étant résolue grâce à un ou plusieurs objets.

On peut en particulier travailler en groupe. Le groupe de programmeurs qui doit résoudre le problème choisit une interface pour chaque objet et répartit ensuite la programmation des différentes classes entre ses membres. Un programmeur donné n'a pas besoin de savoir comment un autre programmeur va *réaliser* les classes dont il est chargé : il lui suffit de savoir comment *utiliser* ces classes, c'est-à-dire quel sens accorder à chaque méthode. Si le découpage est correctement réalisé, ceci permet de répartir harmonieusement la tâche et assure en général une vitesse d'exécution de la programmation plus élevée qu'une autre approche (ainsi qu'une meilleure qualité du résultat final).

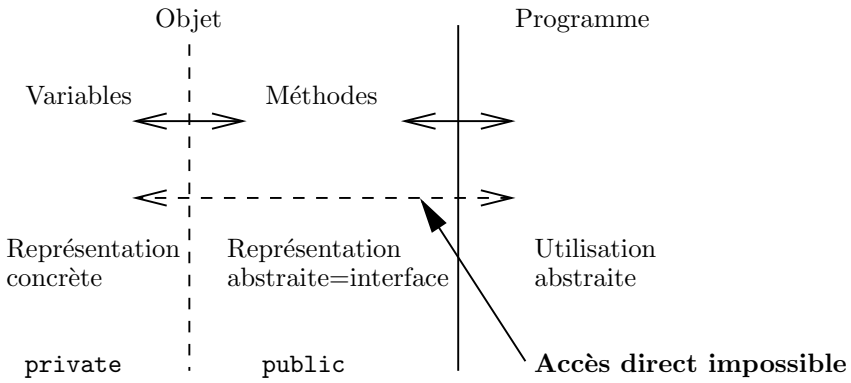

FIG. 9.5 – La notion d'interface dans un objet

La figure 9.5 illustre la notion d'interface ainsi que les aspects concrets et abstraits d'un objet. Notons que seule la connaissance des aspect abstraits et externes d'une classe sont nécessaires à son utilisation. On peut par exemple se douter que `String` et `StringBuffer` sont liés aux tableaux de caractères (`char[]`).

9.4.3 Retour sur `private` et `public`

Les possibilités d'abstraction et de séparation de la programmation de l'utilisation sont souvent basées sur le fait qu'on peut interdire l'accès direct aux variables d'instance d'un objet en dehors

des méthodes de celui-ci.

Dans l'exemple 9.6 qui reprogramme `Complexe` en utilisant un tableau, on a seulement besoin de réécrire la classe elle-même : les programmes qui utilisent cette classe restent complètement inchangés. Pour que ceci soit possible, il faut bien entendu interdire l'accès aux variables d'instance. Si on avait autorisé cet accès, il serait alors indispensable de modifier *tous* les programmes qui utilisent directement une variable d'instance de `Complexe`.

On s'interdira donc en général d'utiliser des variables d'instance accessibles depuis l'extérieur car cette pratique réduit à néant les intérêts des objets décrits dans les sections précédentes. Dans certains cas particuliers (par exemple pour des classes très simples comme `Statistiques`), le recours à des variables d'instance **`public`** peut être très pratique, mais il faudra limiter au maximum ce genre de construction.

9.5 Méthodologie objet

La section précédente nous a permis d'entrevoir les intérêts de l'utilisation des objets. Cependant, nous n'avons pas encore abordé la question des "recettes de cuisine". Comment faire pour construire des objets efficaces et utiles ? Comment choisir les méthodes qui seront le plus utile ? Comment utiliser la puissance d'expression des objets sans se laisser piéger par cette richesse ? Le but de la présente section est de fournir quelques principes informels permettant de construire des objets intéressants.

9.5.1 La conception

La phase de conception des objets d'un programme est la plus importante et la plus difficile. Nous ne prétendons pas donner ici un cours de conception orienté objet, ce qui dépasserait largement le but que nous nous sommes fixé. Nous souhaitons seulement insister sur le fait que la programmation d'une classe doit **toujours** débuter par une phase de conception. Cette phase consiste à se poser deux questions dans un ordre bien précis : d'abord "pourquoi ?", puis "comment ?".

Il s'agit en effet de concevoir une classe qui va remplir un certain rôle. Il est donc important de comprendre et de maîtriser ce rôle. Dans l'exemple évoqué en introduction de ce chapitre, nous avons choisi de travailler sur les nombres complexes. Le rôle de la classe est clair : il s'agit de représenter informatiquement le concept mathématique de nombre complexe. Plus précisément, il s'agit de fournir les outils permettant à un programme de manipuler des nombres complexes, notamment en les utilisant dans des calculs.

Une fois le but défini en quelques phrases, on doit entrer plus précisément dans les détails du "pourquoi ?". Il faut en effet présenter d'abord informellement les **services** que la classe va proposer. Dans l'exemple des nombres complexes, on veut essentiellement pouvoir faire du calcul. Il faut donc pouvoir faire au moins la somme et le produit de deux nombres complexes. Il est aussi utile de pouvoir calculer la norme d'un nombre complexe. D'autres calculs numériques peuvent être utiles (exponentielle complexe, etc.).

Quand la question du "pourquoi ?" est réglée, on obtient un texte introductif et une liste de services. Il faut alors régler la question du "comment ?", elle aussi en plusieurs étapes. Il faut d'abord choisir (comme nous le verrons dans la section 9.5.4) la sémantique générale des instances de la classe : pourra-t-on modifier les instances de la classe ? Dans le cas des complexes, la réponse négative est la plus adaptée.

Ensuite, il faut choisir les méthodes d'instance que la classe proposera (en tenant compte de la sémantique choisie), en fixant leur nom, paramètres et type de retour (après avoir choisi un nom pour la classe). Pour les nombres complexes, nous aurons par exemple besoin d'une méthode `produit` qui prendra en paramètre un complexe et renverra un nouvel objet `Complexe` représentant le produit

du complexe appelant et du complexe paramètre. En résumé, il s'agit de décrire rapidement les méthodes que la classe va proposer, sans pour l'instant décider comment elles seront programmées. C'est la phase de **spécification**. Cette phase peut se réaliser en partie de façon informatique en préparant le texte de la classe, en n'indiquant ni les variables d'instance ni le corps des méthodes. On peut dès ce moment documenter le texte incomplet de la classe, en utilisant la technique décrite à la section 9.5.3. Le texte incomplet ainsi obtenu est appelé **squelette** ou **interface** de la classe.

Exemple 9.7 :

Pour la classe `Complexe`, après l'ajout d'une méthode de calcul du produit, on obtient le squelette documenté suivant (voir la section 9.5.3 pour comprendre les commentaires) :

ComplexeSquel

```
1  /**
2   * La classe Complexe représente informatiquement les nombres complexes.
3   * @author Fabrice Rossi
4   */
5  public class Complexe {
6    /**
7     * Le seul constructeur de la classe Complexe.
8     * @param x la partie réelle
9     * @param y la partie imaginaire
10    */
11   public Complexe(double x,double y) {
12   }
13   /**
14    * Renvoie la partie réelle du complexe appelant.
15    * @return la partie réelle
16    */
17   public double getRéelle() {
18   }
19   /**
20    * Renvoie la partie imaginaire du complexe appelant.
21    * @return la partie imaginaire
22    */
23   public double getImaginaire() {
24   }
25   /**
26    * Renvoie la norme (le module) du complexe appelant.
27    * @return la norme
28    */
29   public double module() {
30   }
31   /**
32    * Renvoie la somme du complexe appelant et du complexe paramètre.
33    * @param c le complexe à ajouter
34    * @return la somme
35    */
36   public Complexe somme(Complexe c) {
37   }
38   /**
39    * Renvoie le produit du complexe appelant et du complexe paramètre.
```

```
40      * @param c le complexe à multiplier
41      * @return le produit
42      */
43     public Complexe produit(Complexe c) {
44     }
45     /**
46      * Transforme le complexe appelant en chaîne de caractères pour pouvoir
47      * l'afficher.
48      * @return la représentation textuelle du complexe
49      */
50     public String toString() {
51     }
52     /**
53      * Compare le complexe appelant au complexe paramètre.
54      * @param c le complexe à comparer au complexe appelant
55      * @return true si et seulement si les deux complexes sont égaux
56      */
57     public boolean equals(Object c) {
58     }
59 }
```

Notons que ce squelette n'est pas complet, car pour obtenir une classe `Complexe` vraiment utile, il faudrait au minimum ajouter des méthodes calculant le quotient de deux complexes, l'argument principal d'un complexe, etc.

La conception est ainsi achevée et on peut maintenant passer à la programmation, phase dans laquelle on répond de façon plus précise à la question du "comment ?", en particulier en choisissant les variables d'instance de la classe et en programmant les différentes méthodes.

9.5.2 Méthodes d'instance utiles

Quand on conçoit une classe, il est difficile de choisir les méthodes d'instance que celle-ci doit fournir afin d'obtenir des instances pratiques à utiliser. C'est pourquoi il est utile de se reporter à la liste suivante qui donne (à titre indicatif) un ensemble minimal de méthodes à fournir :

1. **Constructeurs** :
   Les constructeurs sont bien sûr *quasi*-obligatoires. Il est important de fournir au moins un constructeur pour chaque classe. Il faut bien comprendre que le but du constructeur est de fabriquer un objet directement utilisable. Si on souhaite par exemple représenter des nombres rationnels, sous la forme numérateur/dénominateur, il faut que le constructeur s'arrange pour que le dénominateur ne soit pas nul. Une fois les vérifications effectuées par le constructeur, les autres méthodes peuvent faire l'hypothèse qu'aucun problème le subsiste. En d'autres termes, une fois qu'un objet est créé, on suppose qu'il est correct. Pour l'exemple des rationnels, cela signifie simplement qu'aucune méthode (excepté le constructeur) ne testera si le dénominateur est bien non nul (cf 9.3.4).
2. **Observateurs** :
   Pour manipuler un objet, il faut en général pouvoir accéder aux éléments qu'il représente. Dans le cas des complexes par exemple, il faut pouvoir obtenir la partie réelle et la partie imaginaire du complexe. Il est donc utile d'avoir une méthode qui renvoie la valeur de chacune des variables d'instance de l'objet. Notons que ce n'est absolument pas obligatoire pour deux raisons :

(a) certaines variables peuvent être secrètes : l'utilisateur de l'objet n'a pas à connaître leur valeur (par exemple un mot de passe, ou encore une combinaison comme dans la section 9.6.2) ;

(b) dans certains cas, il plus raisonnable de complètement cacher la structure interne de l'objet et de proposer des méthodes qui donnent un accès *indirect* à son contenu. On peut penser par exemple aux classes `String` et `StringBuffer` dont on ne connaît absolument pas le contenu. Pour un exemple plus concret, voir la classe `Vecteur`, section 9.6.1.

De façon générale, les méthodes qui permettent de connaître le contenu d'un objet sont appelées des **observateurs**.

3. **Conversion en chaîne de caractères** :
   Il quasiment indispensable de pouvoir afficher un objet, la technique la plus simple étant bien entendu de passer par la définition d'une méthode `toString` (cf section 9.2.7).

4. **Test** :
   Comme nous l'avons déjà vu dans l'exemple 9.4 (et comme nous l'étudierons de façon plus complète à la section 9.5.5), il est parfaitement possible de mettre dans une même classe des méthodes d'instance et des méthodes de classe. On peut en profiter pour proposer une méthode `main` qui réalise un test de la classe.

5. **Saisie** :
   On peut aussi mettre à profit les méthodes de classe pour fabriquer des pseudo-constructeurs (voir la section 9.5.5). Une application utile de ceux-ci est la réalisation d'une méthode de saisie qui permet de demander à l'utilisateur la valeur initiale d'un objet. Ceci est particulièrement utile si la saisie de l'objet est complexe et fait intervenir de nombreux éléments.

REMARQUE

La mise en place de saisies dans un constructeur est de façon générale une très mauvaise idée.

9.5.3 La documentation

Quand on construit une classe intéressante, il est indispensable de la **documenter**. Le principe est simple : il s'agit d'utiliser la possibilité d'insérer des **commentaires** dans un programme `Java` (cf la section 1.3.5) afin de donner directement le mode d'emploi de la classe dans le fichier qui la décrit. Le point important est que `Java` propose un format particulier des commentaires qui permet de documenter précisément une classe. On peut ensuite utiliser des programmes spéciaux (`javadoc` et ses dérivés) dont le but est d'extraire les commentaires afin de produire automatiquement une documentation de la classe indépendante du texte de celle-ci. On peut donc lire un texte qui explique comment utiliser une classe sans avoir besoin de lire le fichier qui décrit cette classe (et notamment sans s'occuper de la partie interne de la classe, en particulier les variables d'instance).

Comme nous l'avons dit dans la section 1.3.5, on peut écrire un commentaire de plusieurs lignes en commençant le texte par `/*` et en le terminant par `*/`. Si on commence le texte d'un commentaire par `/**` (on ajoute donc une étoile), `javadoc` reconnaît ce commentaire comme constituant la documentation de la déclaration qui va suivre. Si on place donc ce commentaire avant le début de la description d'une classe, il documente cette classe. Si on le place avant le début de la description d'une méthode, il documente cette méthode, etc. De plus, on peut utiliser des constructions spéciales de la forme `@tag` qui décrivent précisément le rôle de certaines parties de la déclaration qui suit.

Voici comme exemple complet une version documentée de la classe `Complexe` (cette version comporte de nouvelles méthodes que nous présenterons dans les sections suivantes) :

Complexe

```
1  /**
2   * La classe Complexe représente informatiquement les nombres complexes.
3   * @author Fabrice Rossi
4   */
5  public class Complexe {
6    /**
7     * La partie réelle du complexe représenté.
8     */
9    private double re;
10   /**
11    * La partie imaginaire du complexe représenté.
12    */
13   private double im;
14   /**
15    * Le seul constructeur de la classe Complexe.
16    * @param x la partie réelle
17    * @param y la partie imaginaire
18    */
19   public Complexe(double x,double y) {
20     re=x;
21     im=y;
22   }
23   /**
24    * Renvoie la partie réelle du complexe appelant.
25    * @return la partie réelle
26    */
27   public double getRéelle() {
28     return re;
29   }
30   /**
31    * Renvoie la partie imaginaire du complexe appelant.
32    * @return la partie imaginaire
33    */
34   public double getImaginaire() {
35     return im;
36   }
37   /**
38    * Renvoie la norme du complexe appelant.
39    * @return la norme
40    */
41   public double module() {
42     return Math.sqrt(re*re+im*im);
43   }
44   /**
45    * Renvoie la somme du complexe appelant et du complexe paramètre.
46    * @param b le complexe à ajouter
47    * @return la somme
48    */
```

```
49    public Complexe somme(Complexe b) {
50      return new Complexe(re+b.re,im+b.im);
51    }
52    /**
53     * Renvoie le produit du complexe appelant et du complexe paramètre.
54     * @param c le complexe à multiplier
55     * @return le produit
56     */
57    public Complexe produit(Complexe c) {
58      return new Complexe(re*c.re-im*c.im,re*c.im+im*c.re);
59    }
60    /**
61     * Transforme le complexe appelant en chaîne de caractères pour pouvoir
62     * l'afficher.
63     * @return la représentation textuelle du complexe
64     */
65    public String toString() {
66      return "("+re+","+im+")";
67    }
68    /**
69     * Compare le complexe appelant au complexe paramètre.
70     * @param c le complexe à comparer au complexe appelant
71     * @return true si et seulement si les deux complexes sont égaux
72     */
73    public boolean equals(Object c) {
74      if(c instanceof Complexe) {
75        Complexe a=(Complexe)c;
76        return a.re==re && a.im==im;
77      } else {
78        return false;
79      }
80    }
81    /**
82     * Fabrique un représentant complexe d'un nombre réel.
83     * @param x le nombre réel à convertir
84     * @return la représentation complexe
85     */
86    public static Complexe real(double x) {
87      return new Complexe(x,0);
88    }
89    /**
90     * Fabrique un imaginaire pur.
91     * @param x la partie imaginaire du résultat.
92     * @return la représentation complexe
93     */
94    public static Complexe imaginary(double x) {
95      return new Complexe(0,x);
96    }
97    /**
```

```
98      * Méthode de test de la classe Complexe.
99      */
100   public static void main(String[] args) {
101     Complexe a=new Complexe(2.0,3.0);
102     /* test du constructeur et des observateurs */
103     System.out.println(a.getRéelle());
104     System.out.println(a.getImaginaire());
105     System.out.println(a);
106     Complexe b=new Complexe(-2.5,5.5);
107     System.out.println(b);
108     /* test de la somme */
109     Complexe c=a.somme(b);
110     System.out.println(a);
111     System.out.println(b);
112     System.out.println(c);
113     /* test de la norme */
114     c=new Complexe(0.5,-5);
115     System.out.println(c.module());
116   }
117 }
```

On remarque que chaque élément (la classe, les variables et les méthodes) est documenté. De plus, chaque ligne de documentation est précédée par une étoile alignée avec la première étoile du commentaire de documentation. Ceci permet de repérer facilement les documentations dans le texte du programme.

Expliquons maintenant le rôle des constructions spéciales qui apparaissent dans les commentaires de documentation :

`@author` `nom`

Permet d'indiquer qui est l'auteur de la classe.

`@param` `identificateur` `texte`

Permet de documenter un paramètre d'une méthode. L'identificateur correspondant au paramètre est donné après la construction `@param`.

`@return` `texte`

Permet de documenter le résultat renvoyé par une méthode.

Les figures 9.6 à 9.9 présentent la documentation après transformation par l'outil `javadoc` et telle qu'elle serait visualisée au moyen d'un navigateur *Web*[6]. La documentation correspond à la classe complète, obtenue après ajout des méthodes de classe étudiées à la section 9.5.5. Dans l'affichage choisit, on trouve la description des variables d'instance. Dans la pratique, on peut choisir de cacher tout ce qui est déclaré comme `private`, ce qui est une bonne pratique. En effet, la documentation d'une classe est justement destinée à fournir une description de son utilisation, en masquant les détails techniques de sa programmation : les variables d'instance `private` sont justement des détails de programmation.

9.5.4 Deux sémantiques : immuable et modifiable

Comme nous l'avons évoqué à la section 9.5.1, la conception d'une classe impose de choisir la **sémantique** qui sera associée aux instances de celle-ci. Doit-on pouvoir modifier les objets ou non ?

[6] Comme par exemple le logiciel `Communicator` de la société `Netscape`, cf. `http://www.netscape.com`

Class Deprecated

PREV CLASS NEXT CLASS | **FRAMES NO FRAMES**
SUMMARY: INNER | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

Class Complexe

```
java.lang.Object
  |
  +--Complexe
```

public class **Complexe**
extends java.lang.Object

La classe Complexe représente informatiquement les nombres complexes.

Author:
Fabrice Rossi

Constructor Summary

Complexe(double x, double y) Le seul constructeur de la classe Complexe.

Method Summary

boolean	**equals**(java.lang.Object c) Compare le complexe appelant au complexe paramètre.
double	**getImaginaire**() Renvoie la partie imaginaire du complexe appelant.
double	**getRéelle**() Renvoie la partie réelle du complexe appelant.
static Complexe	**imaginary**(double x) Fabrique un imaginaire pur.
static void	**main**(java.lang.String[] args) Méthode de test de la classe Complexe.
double	**module**() Renvoie la norme du complexe appelant.
Complexe	**produit**(Complexe c) Renvoie le produit du complexe appelant et du complexe paramètre.

FIG. 9.6 – Documentation de la classe `Complexe`

Method Summary (continued)

static Complexe	**real**(double x) Fabrique un représentant complexe d'un nombre réel.
Complexe	**somme**(Complexe b) Renvoie la somme du complexe appelant et du complexe paramètre.
java.lang.String	**toString**() Transforme le complexe appelant en chaîne de caractères pour pouvoir l'afficher.

Methods inherited from class java.lang.Object
clone, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Complexe

```
public Complexe(double x,
                double y)
```

Le seul constructeur de la classe Complexe.
Parameters:
x - la partie réelle
y - la partie imaginaire

Method Detail

getRéelle

```
public double getRéelle()
```

Renvoie la partie réelle du complexe appelant.
Returns:
la partie réelle

---

getImaginaire

```
public double getImaginaire()
```

Renvoie la partie imaginaire du complexe appelant.
Returns:
la partie imaginaire

FIG. 9.7 – Documentation de la classe `Complexe`

module

```
public double module()
```

Renvoie la norme du complexe appelant.
Returns:
la norme

---

somme

```
public Complexe somme(Complexe b)
```

Renvoie la somme du complexe appelant et du complexe paramètre.
Parameters:
`b` - le complexe à ajouter
Returns:
la somme

---

produit

```
public Complexe produit(Complexe c)
```

Renvoie le produit du complexe appelant et du complexe paramètre.
Parameters:
`c` - le complexe à multiplier
Returns:
le produit

---

toString

```
public java.lang.String toString()
```

Transforme le complexe appelant en chaîne de caractères pour pouvoir l'afficher.
Overrides:
toString in class java.lang.Object
Returns:
la représentation textuelle du complexe

---

equals

```
public boolean equals(java.lang.Object c)
```

Compare le complexe appelant au complexe paramètre.
Overrides:

FIG. 9.8 – Documentation de la classe `Complexe`

equals in class java.lang.Object

Parameters:

c - le complexe à comparer au complexe appelant

Returns:

true si et seulement si les deux complexes sont égaux

---

real

```
public static Complexe real(double x)
```

Fabrique un représentant complexe d'un nombre réel.

Parameters:

x - le nombre réel à convertir

Returns:

la représentation complexe

---

imaginary

```
public static Complexe imaginary(double x)
```

Fabrique un imaginaire pur.

Parameters:

x - la partie imaginaire du résultat.

Returns:

la représentation complexe

---

main

```
public static void main(java.lang.String[] args)
```

Méthode de test de la classe Complexe.

Class Deprecated

PREV CLASS NEXT CLASS — **FRAMES NO FRAMES**

SUMMARY: INNER | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

FIG. 9.9 – Documentation de la classe `Complexe`

Sémantique immuable

On peut d'abord choisir pour une classe la sémantique immuable. Le principe est qu'une fois un objet créé, il est impossible de modifier son état (encore appelé contenu). L'exemple de base est celui de la classe `String`. Chaque objet `String` représente une chaîne de caractères absolument impossible à modifier.

Pour qu'une classe possède une sémantique immuable, il faut qu'aucune de ses méthodes d'instance ne permette de modifier l'objet appelant. Toute opération de modification doit produire un résultat représenté par un nouvel objet, distinct de l'objet appelant. Il est ainsi possible de mettre bout à bout deux chaînes de caractères, mais le résultat est un nouvel objet `String` et les objets de départ sont restés intacts.

L'intérêt d'une classe immuable est sa facilité d'utilisation : aucun effet de bord n'est possible et d'un point de vue extérieur, une instance d'une telle classe se comporte exactement comme une valeur d'un type fondamental.

L'inconvénient principal d'une classe immuable est l'efficacité réduite : chaque modification provoque la création d'un nouvel objet, ce qui peut être très long.

Sémantique modifiable

Une classe possédant une sémantique modifiable permet de produire des objets dont l'état est modifiable grâce aux méthodes d'instance. En général, une telle classe ne propose pas de méthode d'instance permettant de construire un nouvel objet représentant le résultat d'une modification de l'objet appelant. Le seul moyen d'arriver à ce résultat est de commencer par recopier l'objet appelant (avec une méthode appelée en général `clone`) puis d'appeler la méthode de modification sur ce nouvel objet, indépendant du premier (mais représentant au départ la même valeur). L'exemple de base d'une classe à sémantique modifiable est la classe `StringBuffer`.

Le principal intérêt de ce type de classe est l'efficacité : on crée peu d'objets et on modifie ceux-ci afin d'éviter de recopier des informations.

Le défaut majeur est bien sûr la difficulté d'utilisation à cause des effets de bord possibles.

Comment choisir ?

C'est un problème très délicat. Il faut mettre en balance les avantages et les inconvénients. Par exemple pour les nombres complexes, le but principal est le calcul numérique dans $\mathbb{C}$. Or, la présence d'effet de bord est assez éloignée de l'intuition mathématique et la sémantique immuable semble donc la bienvenue, d'autant plus qu'un complexe est un "petit" objet et qu'une recopie ne prendra pas trop de temps.

Si on souhaite mettre en place une classe permettant de gérer des tableaux dynamiques (quelque chose ressemblant aux tableaux classiques, mais avec la possibilité comme dans les `StringBuffer` d'allonger ou de réduire la taille du tableau), il semble au contraire plus intéressant de choisir une sémantique modifiable. En effet, si on manipule des tableaux avec un nombre important d'éléments, il est hors de question de devoir recopier le contenu intégral de l'un d'eux afin de produire le résultat d'une modification. De plus, il n'y a pas ici d'abstraction mathématique à représenter et on peut demander à l'utilisateur d'être un peu plus attentif aux détails informatiques et donc aux effets de bord.

Notons qu'il est possible d'utiliser une sémantique hybride (qui mélange les sémantiques immuable et modifiable), à condition de bien justifier ce choix. En effet, dans certains cas, il est important de pouvoir modifier efficacement un objet car il serait trop coûteux de produire un nouvel objet à chaque modification. Mais pour les mêmes cas, il peut aussi être important de proposer

des méthodes qui réalisent des calculs sans modifier l'objet appelant, en particulier pour se conformer à l'intuition mathématique. Nous verrons un exemple de classe hybride dans la section 9.6.1 quand nous présenterons une classe `Vecteur`.

Enfin, il est important de garder à l'esprit que les objets peuvent servir à modéliser des entités du monde réel, dont l'état peut en général être modifié. On verra à la section 9.6.2 un tel exemple pour lequel la sémantique modifiable est la seule envisageable.

9.5.5 Les méthodes de classe

Nous savons maintenant écrire à la fois des méthodes d'instance et des méthodes de classe. Est-il alors légitime de mélanger dans une même classe des méthodes de natures différentes ?

C'est en effet une bonne idée, essentiellement pour deux raisons :

1. **Le test**

   Il est très important de vérifier qu'une classe fonctionne car on peut alors l'utiliser dans un autre programme en étant à peu près sûr que les éventuels problèmes viendront du nouveau programme et pas de la classe. Pour ce faire, on réalise un programme de test. Un tel programme doit essayer toutes les méthodes de la classe, avec différents arguments, et afficher les paramètres et les résultats afin que l'utilisateur puisse vérifier qu'ils correspondent bien à ceux attendus[7].

   Au lieu d'écrire le programme de test dans une classe indépendante de la classe développée, nous ajouterons une méthode `main` à la classe. Il sera alors possible de lancer le programme défini par cette classe, c'est-à-dire l'exécution de la méthode `main`.

   Pour la classe `Complexe`, déjà longuement étudiée, on peut utiliser la méthode `main` suivante :

main

```
1  /**
2   * Méthode de test de la classe Complexe.
3   */
4  public static void main(String[] args) {
5    Complexe a=new Complexe(2.0,3.0);
6    /* test du constructeur et des observateurs */
7    System.out.println(a.getRéelle());
8    System.out.println(a.getImaginaire());
9    System.out.println(a);
10   Complexe b=new Complexe(-2.5,5.5);
11   System.out.println(b);
12   /* test de la somme */
13   Complexe c=a.somme(b);
14   System.out.println(a);
15   System.out.println(b);
16   System.out.println(c);
17   /* test de la norme */
18   c=new Complexe(0.5,-5);
19   System.out.println(c.module());
20 }
```

2. **Pour les classes immuables**

[7] il est souvent utile de faire tester par le programme les résultats obtenus, en écrivant directement dans le `main` les résultats qu'on devrait théoriquement obtenir.

Dans une classe possédant une sémantique immuable, il est parfois utile de fournir des services permettant de créer une instance de la classe sans passer par un constructeur. En effet, les constructeurs doivent impérativement porter le nom de la classe et différer entre eux par le nombre de paramètres et/ou les types des paramètres (voir la section 9.2.6). Ceci constitue parfois une limitation.

Considérons en effet l'exemple des complexes. Si on souhaite créer un nombre réel ou un imaginaire pur, on est obligé de transmettre au constructeur général un paramètre nul (sans se tromper d'ordre !). On ne peut pas créer deux constructeurs adaptés car ils prendraient chacun un unique paramètre réel. Le seul moyen de simplifier la création est donc de fournir deux "pseudo constructeurs". Il est impossible d'utiliser des méthodes d'instance car elles travaillent sur un objet déjà créé et notre classe est immuable. Donc, on doit utiliser des méthodes de classe. On ajoute alors à la classe `Complexe` les méthodes suivantes :

ComplexeStatic

```
1  /**
2   * Fabrique un représentant complexe d'un nombre réel.
3   * @param x le nombre réel à convertir
4   * @return la représentation complexe
5   */
6  public static Complexe real(double x) {
7    return new Complexe(x,0);
8  }
9  /**
10  * Fabrique un imaginaire pur.
11  * @param x la partie imaginaire du résultat.
12  * @return la représentation complexe
13  */
14 public static Complexe imaginary(double x) {
15   return new Complexe(0,x);
16 }
17
```

L'utilisation d'une telle méthode est très simple. Il suffira par exemple d'écrire :

```
Complexe c=Complexe.imaginary(2.5) ;
```

Nous avons déjà vu un exemple de ce genre de méthode dans la section 7.1.4. La classe `String` propose en effet des méthodes `valueOf` qui s'utilisent en écrivant par exemple :

```
String t=String.valueOf(5.5) ;
```

Ces méthodes de classe permettent de créer une chaîne de caractères représentant une valeur numérique.

REMARQUE

Il est très important de conserver à l'esprit qu'une méthode de classe écrite dans une classe instanciable ne peut pas manipuler les variables d'instance directement (comme le ferait une méthode d'instance). Dans une méthode de classe de la classe `Complexe`, il est par exemple impossible d'écrire `re=0 ;`. En effet, cette écriture est possible dans une méthode d'instance seulement car l'objet appelant est un paramètre *implicite* de la méthode et que l'instruction `re=0 ;` peut se traduire par `this.re=0 ;`.

9.6 Exemples

9.6.1 Les vecteurs de $\mathbb{R}^n$

Conception

Nous cherchons ici à produire une classe `Vecteur` qui doit représenter informatiquement les éléments de $\mathbb{R}^n$. Nous souhaitons que cette classe propose comme service les manipulations élémentaires de vecteurs :

- somme de deux vecteurs ;
- multiplication par un réel ;
- produit scalaire de deux vecteurs ;
- norme d'un vecteur.

Considérons maintenant le problème de la sémantique de la classe `Vecteur`. Pour pouvoir manipuler un vecteur, il faut d'abord le créer. Imaginons qu'après la création, on ne puisse plus changer la valeur de l'objet obtenu. Pour pouvoir engendrer n'importe quel vecteur, il faut donc que le constructeur permette de donner la valeur de chaque coordonnée du vecteur. Ceci est envisageable en prévoyant un constructeur qui accepte comme paramètre un tableau de réels. On écrira alors une construction de la façon suivante :

```
Vecteur v=new Vecteur(new double[] {1.0,0.0,0.0});
```

Cette instruction devrait permettre par exemple de créer la représentation informatique du vecteur $(1, 0, 0)$.

Ce choix de la sémantique immuable pose cependant un problème : comment faire pour créer un vecteur dont les coordonnées vont être calculées par un programme ? Avec la méthode retenue, le plus simple est de d'abord créer un tableau (de type `double[]`), placer dedans les valeurs numériques voulues, puis de créer le vecteur avec comme paramètre ce tableau. Cette approche enlève en partie l'intérêt de l'approche objet : on peut manipuler abstraitement un vecteur, mais il faut d'abord l'avoir créé par une méthode on ne peut moins concrète.

Une solution envisageable est d'avoir une sémantique hybride et en particulier de permettre la modification des coordonnées d'un vecteur. L'idée est de proposer une méthode permettant de changer la valeur de la k-ième coordonnée d'un vecteur en modifiant l'objet, exactement comme la méthode `setCharAt` des `StringBuffer` permet la modification d'un caractère dans une chaîne directement au niveau de l'objet.

Spécification

Passons maintenant à l'étape de spécification qui précise quelles méthodes nous allons définir dans la classe `Vecteur`. Notons que pour être cohérent avec les notations mathématiques, les coordonnées des vecteurs seront numérotées de 1 à la dimension de l'espace.

`Vecteur(int n)`

Ce constructeur permet de créer un `Vecteur` de $\mathbb{R}^{\mathtt{n}}$. Ses composantes sont initialisées à zéro.

`Vecteur(double[] t)`

Ce constructeur transforme un tableau de réels en un `Vecteur`. On s'arrangera pour que l'objet obtenu soit complètement indépendant du tableau.

`int getDimension()`

Cette méthode renvoie la dimension de l'espace vectoriel auquel le vecteur appelant appartient.

`Vecteur setCoordinate(int k,double x)`

Cette méthode permet de changer la valeur de la `k`-ième coordonnée du vecteur appelant en lui donnant la valeur `x`. La méthode renvoie une référence sur l'objet appelant.

`double getCoordinate(int k)`

Cette méthode renvoie la valeur de la `k`-ième coordonnée du vecteur appelant.

`Vecteur add(Vecteur b)`

Cette méthode renvoie un nouvel objet `Vecteur` somme du vecteur appelant et du vecteur paramètre `b`.

`Vecteur mult(double x)`

Cette méthode renvoie un nouvel objet `Vecteur` égal au vecteur appelant multiplié par le réel `x`.

`double scalar(Vector b)`

Cette méthode renvoie le produit scalaire du vecteur appelant et du vecteur `b`.

`double norm()`

Cette méthode renvoie la norme euclidienne du vecteur appelant.

`String toString()`

Cette méthode renvoie une chaîne de caractères représentant le vecteur appelant.

`boolean equals(Object o)`

Cette méthode compare le vecteur appelant avec le vecteur paramètre, renvoyant `true` en cas d'égalité des contenus.

Cette spécification permet d'écrire le squelette suivant :

VecteurSquel

```
1  public class Vecteur {
2    public Vecteur(int n) {
3    }
4    public Vecteur(double[] t) {
5    }
6    public int getDimension() {
7    }
8    public Vecteur setCoordinate(int k,double x) {
9    }
10   public double getCoordinate(int k) {
11   }
12   public Vecteur add(Vecteur b) {
13   }
14   public Vecteur mult(double x) {
15   }
16   public double scalar(Vecteur b) {
17   }
18   public double norm() {
19   }
20   public String toString() {
21   }
22   public boolean equals(Object c) {
23   }
24 }
```

Programmation

Il est naturel de représenter un vecteur de $\mathbb{R}^n$ par un tableau de réels. Le seul point "délicat" est qu'il faut traduire les coordonnées mathématiques en numéro d'indice Java. En effet, nous avons choisi d'utiliser pour les vecteurs la notation mathématique, avec les coordonnées numérotées à partir de 1.

Voici le texte de la classe `Vecteur` :

Vecteur

```
1  /**
2   * La classe Vecteur représente informatiquement les vecteurs de R^n.
3   * @author Fabrice Rossi
4   */
5  public class Vecteur {
6    /**
7     * les coordonnées du vecteur sont stockées dans ce tableau.
8     */
9    private double[] valeur;
10   /**
11    * Crée un vecteur de dimension n dont toutes les coordonnées sont nulles.
12    * @param n dimension
13    */
14   public Vecteur(int n) {
15     // on crée un tableau à n cases. Les cases sont automatiquement
16     // initialisées à 0, ce qui est exactement ce que l'on veut.
17     valeur=new double[n];
18   }
19   /**
20    * Crée un vecteur dont les coordonnées sont données par le contenu du
21    * tableau passé en paramètre.
22    * @param t coordonnées du vecteur à construire
23    */
24   public Vecteur(double[] t) {
25     // Attention ! Il faut recopier le contenu du tableau, sinon on pourra
26     // modifier le vecteur depuis l'exterieur.
27     if (t!=null)
28       valeur=(double[])t.clone();
29   }
30   /**
31    * Renvoie la dimension du vecteur appelant.
32    * @return la dimension
33    */
34   public int getDimension() {
35     return valeur.length;
36   }
37   /**
38    * Modifie une coordonnée du vecteur appelant.
39    * @param k numéro de la coordonnée
40    * @param x nouvelle valeur
41    * @return le vecteur appelant
42    */
```

```
43    public Vecteur setCoordinate(int k,double x) {
44      valeur[k-1]=x;
45      return this;
46    }
47    /**
48     * Renvoie la valeur d'une coordonnée du vecteur appelant.
49     * @param k numéro de la coordonnée
50     * @return la valeur de la coordonnée
51     */
52    public double getCoordinate(int k) {
53      return valeur[k-1];
54    }
55    /**
56     * Renvoie un nouveau vecteur somme du vecteur appelant et du vecteur
57     * paramètre.
58     * @param b deuxième opérande de la somme
59     * @return le vecteur somme
60     */
61    public Vecteur add(Vecteur b) {
62      Vecteur result=new Vecteur(valeur.length);
63      for(int i=0;i<valeur.length;i++)
64        result.valeur[i]=valeur[i]+b.valeur[i];
65      return result;
66    }
67    /**
68     * Renvoie un nouveau vecteur obtenu en multipliant par un réel le vecteur
69     * appelant.
70     * @param x le scalaire
71     * @return le vecteur résultat
72     */
73    public Vecteur mult(double x) {
74      Vecteur result=new Vecteur(valeur.length);
75      for(int i=0;i<valeur.length;i++)
76        result.valeur[i]=x*valeur[i];
77      return result;
78    }
79    /**
80     * Renvoie le produit scalaire du vecteur appelant et d'un vecteur
81     * paramètre.
82     * @param b le second vecteur
83     * @return le produit scalaire
84     */
85    public double scalar(Vecteur b) {
86      double result=0;
87      for(int i=0;i<valeur.length;i++)
88        result+=valeur[i]*b.valeur[i];
89      return result;
90    }
91    /**
```

```
92    * Renvoie la norme du vecteur appelant.
93    * @return la norme
94    */
95   public double norm() {
96     return Math.sqrt(scalar(this));
97   }
98   /**
99    * Renvoie la représentation sous forme de String du vecteur appelant.
100   * @return le vecteur sous forme de chaîne de caractères.
101   */
102  public String toString() {
103    StringBuffer preresult=new StringBuffer("(");
104    for(int i=0;i<valeur.length;i++)
105      preresult.append(String.valueOf(valeur[i])).append(',');
106    preresult.setCharAt(preresult.length()-1,')');
107    return preresult.toString();
108  }
109  /**
110   * Compare le vecteur appelant au vecteur paramètre.
111   * @param o le vecteur à comparer au vecteur appelant
112   * @return true si et seulement si les deux vecteurs sont égaux
113   */
114  public boolean equals(Object o) {
115    if(o==this)
116      return true;
117    if(o instanceof Vecteur) {
118      Vecteur v=(Vecteur)o;
119      if(v.valeur.length!=valeur.length)
120        return false;
121      for(int i=0;i<valeur.length;i++)
122        if(v.valeur[i]!=valeur[i])
123          return false;
124      return true;
125    } else {
126      return false;
127    }
128  }
129  /**
130   * methode de test
131   */
132  public static void main(String[] args) {
133    double[] temp={1,2,3};
134    Vecteur a=new Vecteur(temp);
135    System.out.println(a);
136    System.out.println(a.norm());
137    System.out.println(a.getDimension());
138    for(int i=1;i<=a.getDimension();i++)
139      System.out.println(String.valueOf(i)+"->"+a.getCoordinate(i));
140    Vecteur b=new Vecteur(3);
```

```
141     System.out.println(b);
142     b.setCoordinate(1,2);
143     System.out.println(b);
144     Vecteur c=a.add(b);
145     System.out.println(c);
146     System.out.println(c.scalar(b));
147     c=c.mult(2.5);
148     System.out.println(c);
149     Vecteur d=new Vecteur(new double[] {7.5,5,7.5});
150     System.out.println(d.equals(d));
151     System.out.println(d.equals(c));
152     System.out.println(d.equals(b));
153     Vecteur e=new Vecteur(1);
154     System.out.println(d.equals(e));
155   }
156 }
```

Certaines méthodes de cette classe demandent quelques explications :

- Le constructeur `Vecteur(double[] t)` est assez complexe car il doit recopier le contenu du tableau paramètre dans un nouveau tableau. On peut se demander pourquoi. En fait, la réponse est simple.
  Imaginons qu'on ne copie pas le tableau, ce qui donne le constructeur suivant :

  VecteurConstructeur

```
1 public Vecteur(double[] t) {
2   valeur=t;
3 }
```

  Alors, comme les tableaux sont manipulés par référence, on peut provoquer des effets de bord, c'est-à-dire modifier un vecteur grâce à un tableau, comme le montre l'exemple (hypothétique) suivant :

```
double[] t=new double[1];
t[0]=2;
Vecteur a=new Vecteur(t);
t[0]=3;
System.out.println(a.getCoordinate(1));
```

  La dernière ligne du programme afficherait alors `3`, car le tableau contenu dans le vecteur et le tableau auquel `t` fait référence seraient en fait un seul et unique tableau, comme l'illustre la figure 9.10.
- La méthode `equals` est légèrement différente des méthodes proposées dans les classes précédentes, et de la forme générale donnée à la section 9.2.8. En effet, elle commence par une ligne qui compare `this` avec le paramètre de la méthode. Cette technique est une simple astuce de programmation qui permet de traiter efficacement le cas où on compare un objet avec lui-même (cas pour lequel on a donc égalité entre `this` et le paramètre). En effet, le test d'égalité pour les vecteurs est relativement long : on doit comparer le contenu complet des vecteurs avant de pouvoir conclure. Pour les dimensions élevées, ceci peut être relativement lent.
- La méthode `setCoordinate` introduit une construction que nous n'avons pas encore étudiée : **`return this`**. Techniquement, il n'y a rien de nouveau ici : la méthode renvoie la référence

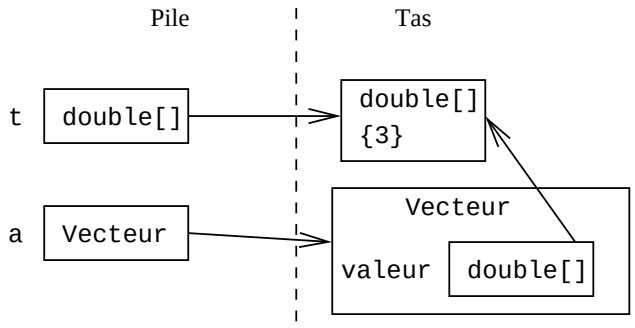

FIG. 9.10 – Effet de bord quand on ne copie pas le tableau

contenue dans la variable `this`, c'est-à-dire la référence sur l'objet appelant. L'intérêt de cette action est de permettre l'enchaînement d'appels. Considérons l'exemple suivant :

```
Vecteur v=new Vecteur(3);
v.setCoordinate(1,2.5).setCoordinate(3,-2.8);
System.out.println(v);
```

Ce programme est parfaitement accepté par le compilateur et produit l'affichage suivant :

AFFICHAGE

```
(2.5,0.0,-2.8)
```

On peut alors se demander comment la deuxième ligne du programme est interprétée. En fait, c'est relativement simple. L'ordinateur exécute les appels de méthode de gauche à droite. Donc, pour lui, la ligne deux est équivalente à :

```
(v.setCoordinate(1,2.5)).setCoordinate(3,-2.8);
```

Il commence par exécuter `v.setCoordinate(1,2.5)`, ce qui a pour effet de modifier l'objet appelant en fixant la coordonnée numéro 1 à 2.5. De plus, la méthode renvoie une référence sur l'objet appelant, ce qui veut dire que l'appel est remplacé par cet référence. De ce fait, *après l'exécution du premier appel*, tout se passe comme si on avait maintenant à exécuter :

```
v.setCoordinate(3,-2.8);
```

Cette instruction a pour effet de modifier l'objet appelant en fixant la coordonnée numéro 3 à -2.8, d'où le résultat final obtenu.

REMARQUE

La classe proposée ici est incomplète car elle ne tient pas du tout compte des éventuelles erreurs. Il est parfaitement possible par exemple de demander le produit scalaire de deux vecteurs de dimensions différentes, ce qui devrait provoquer l'affichage d'un message d'erreur. Comme nous n'avons pas vu jusqu'à présent comment traiter de façon efficace les erreurs, nous nous contenterons de cette version des `Vecteur`s.

9.6.2 Un objet `Serrure`

Conception

La section précédente présentait un exemple très mathématique, comme d'ailleurs l'exemple des `Complexe`s qui a servi de fil conducteur à ce chapitre. Nous souhaitons donner ici un exemple

plus pragmatique montrant que les objets peuvent servir à modéliser le réel sans passer par une représentation mathématique.

Il s'agit ici de représenter une serrure à code. Une telle serrure peut être soit ouverte, soit fermée. La fermeture est simple, mais l'ouverture ne l'est pas. On suppose qu'il s'agit d'une serrure de coffre-fort : la combinaison est constituée de trois entiers relatifs. Un entier positif indique une rotation dans le sens trigonométrique ("vers la gauche"), alors qu'un entier négatif représente une rotation dans le sens contraire (donc, "vers la droite"). De plus, il y a obligatoirement changement de sens de rotation après chaque rotation : cela signifie qu'une rotation vers la gauche est nécessairement suivie d'une rotation vers droite, elle-même suivie d'une rotation vers la gauche. Il y a une seule possibilité : commencer par la rotation vers la droite. La combinaison $(4, 3, -5)$ n'est pas correcte : on doit tourner deux fois de suite vers la gauche. Par contre la combinaison $(2, -3, 4)$ est correcte et se traduit dans le langage courant par : 2 crans à gauche, 3 crans à droite et 4 crans à gauche. La serrure devra donc fournir les services suivants :

- fabrication de la serrure, avec choix de la combinaison (non modifiable par la suite !) ;
- rotation à droite et à gauche ;
- remise à zéro de la molette (la pièce qu'on fait tourner) ;
- fermeture de la serrure.

Le comportement visé est le suivant : si la serrure est fermée et qu'on applique les bonnes rotations (dans le bon ordre), la serrure s'ouvre. Ceci n'est pas aussi simple qu'on pourrait le croire, comme nous le verrons lors de la programmation.

Spécification

Passons maintenant à l'étape de spécification qui précise les méthode que nous allons définir pour la classe `Serrure` :

`Serrure(int a,int b,int c)`

> Ce constructeur permet de créer une serrure en fixant comme combinaison le code `(a,b,c)`. Le constructeur fait les vérifications de cohérence sur la combinaison. Il affiche un message d'erreur en cas de combinaison incorrecte et fixe alors la combinaison $(0, 0, 0)$. La serrure est initialement créée ouverte.

`Serrure tourne(int crans)`

> Tourne la molette de $|$`crans`$|$ crans vers la droite si `crans` est négatif, vers la gauche dans le cas contraire. Renvoie une référence vers l'objet appelant.

`Serrure zéro()`

> Place la molette de la serrure appelante en position zéro et renvoie une référence vers l'objet appelant.

`boolean estOuverte()`

> Renvoie `true` si et seulement si la serrure est ouverte.

`void ferme()`

> Ferme la serrure.

Programmation

Il nous reste maintenant à traiter le problème assez délicat de la programmation de la classe `Serrure`. Voici une analyse conduisant à une solution possible :

- il est d'abord clair que la `Serrure` doit posséder trois variables d'instance contenant la combinaison ;

- il est clair aussi que la `Serrure` doit posséder une variable d'instance de type `boolean` indiquant si elle est ouverte ou non ;
- la partie délicate concerne les opérations de rotation. Grâce à une variable d'instance, on peut connaître la position actuelle de la molette (obtenue par un entier relatif). Cependant, cette information n'est pas suffisante. En effet, il faut effectuer les rotations dans l'ordre de la combinaison pour ouvrir la serrure. De ce fait, c'est la **séquence** des positions qui importe. Plus précisément, si la combinaison est (a, b, c), alors il faut positionner d'abord la molette en zéro, puis en a, en $a+b$ et pour finir en $a+b+c$ pour ouvrir la serrure (car les rotations sont relatives). Donc, en fait, il faut conserver les quatre dernières positions de la molette pour pouvoir déterminer si la serrure a été ouverte.

Voici le texte de la classe `Serrure` :

Serrure

```
1  /**
2   * La classe Serrure représente informatiquement une serrure de coffre-fort
3   * @author Fabrice Rossi
4   */
5  public class Serrure {
6    /**
7     * stockage de la suite des positions d'ouverture
8     */
9    private int[] combinaison;
10   /**
11    * stockage de la suite des dernières positions
12    */
13   private int[] positions;
14   /**
15    * stockage de la combinaison
16    */
17   private int premier,deuxième,troisième;
18   /**
19     * état de la serrure (ouverte ou fermée)
20     */
21   private boolean ouverte;
22   /**
23    * Crée une serrure dont la combinaison est fixée en paramètre
24    * @param a première rotation de la combinaison
25    * @param b deuxième rotation de la combinaison
26    * @param c troisième rotation de la combinaison
27    */
28   public Serrure(int a,int b,int c) {
29     // vérifications
30     if (!( (a>0 && b<0 && c>0) || (a<0 && b>0 && c<0) )) {
31       System.out.println("Erreur dans la combinaison : "+a+","+b+","+c);
32       a=0;
33       b=0;
34       c=0;
35     }
36     premier=a;
37     deuxième=b;
```

```
38      troisième=c;
39      positions=new int[4];
40      combinaison=new int[] {0,a,a+b,a+b+c};
41      ouverte=true;
42    }
43    /**
44     * Décale d'un rang vers la gauche le tableau des positions : supprime la
45     * première et remplace la dernière par la position passée en paramètre.
46     * @param newPos la nouvelle position.
47     */
48    private void décale(int newPos) {
49      // décalage des positions
50      for(int i=1;i<positions.length;i++)
51        positions[i-1]=positions[i];
52      positions[positions.length-1]=newPos;
53    }
54    /**
55     * Effectue une rotation de la serrure.
56     * @param crans nombre de crans pour la rotation, sens indiqué par le signe.
57     * @return l'objet appelant
58     */
59    public Serrure tourne(int crans) {
60      // nouvelle position
61      décale(positions[positions.length-1]+crans);
62      // comparaison
63      ouverte=true;
64      for(int i=1;i<positions.length;i++)
65        if (positions[i]!=combinaison[i]) {
66          ouverte=false;
67          break;
68        }
69      return this;
70    }
71    /**
72     * Positionne à zéro la molette de la serrure.
73     * @return l'objet appelant
74     */
75    public Serrure zéro() {
76      décale(0);
77      return this;
78    }
79    /**
80     * Renvoie l'état de la serrure.
81     * @return true pour une serrure ouverte, false sinon.
82     */
83    public boolean estOuverte() {
84      return ouverte;
85    }
86    /**
```

```
87      * Ferme la serrure appelante.
88      */
89     public void ferme() {
90       ouverte=false;
91     }
92     /**
93      * methode de test
94      */
95     public static void main(String[] args) {
96       Serrure s=new Serrure(-2,3,-4);
97       s.ferme();
98       // Essai d'une combinaison fausse, mais presque juste
99       s.zéro();
100      s.tourne(-2);
101      s.tourne(2);
102      s.tourne(-4);
103      System.out.println(s.estOuverte());
104      // Nouvel essai, avec un début faux
105      s.zéro();
106      s.tourne(4);
107      // on revient finalement en arrière
108      s.tourne(-4); // ceci repositionne en 0
109      s.tourne(-2);
110      s.tourne(3);
111      s.tourne(-4);
112      // et hop
113      System.out.println(s.estOuverte());
114    }
115 }
```

Ce texte demande quelques éclaircissements :

- le tableau `combinaison` contient la suite des positions de la molette qui permet d'ouvrir la serrure ;
- le tableau `positions` contient la suite des quatre dernières positions de la molette. A chaque fois qu'on tourne la molette (cf les méthodes `tourne` et `zéro`), on décale les positions : on perd la première du tableau (qui est la plus ancienne) et on ajoute dans la dernière case du tableau la nouvelle position. Tout ceci est effectué par la méthode `décale` qui est une méthode d'instance privée ;
- pour savoir si la serrure est ouverte, on compare les dernières positions de la molette avec la suite de positions qui assure l'ouverture : c'est le propos de la seconde boucle de la méthode `tourne` qui utilise une évaluation court-circuitée.

9.7 Conseils d'apprentissage

Comme nous l'avons déjà dit au chapitre 7, il n'est pas possible de programmer de façon vraiment utile en `Java` sans utiliser d'objets. Il est tout aussi indispensable de programmer ses propres objets pour pouvoir développer de réels programmes `Java` (pour certaines applications comme le graphisme que nous étudierons au chapitre 10), il est même strictement impossible de se passer de la programmation d'objets).

Fort heureusement, nous avons déjà abordé à partir du chapitre 7 les réelles difficultés techniques de la manipulation des objets, à savoir la manipulation par référence. Même si le présent chapitre contient quelques nouveautés (essentiellement la définition de variables et de méthodes d'instance), il n'y a pas de réelle difficulté supplémentaire au niveau des mécanismes. En fait, le point le plus délicat de la programmation objet est méthodologique : comment faire pour mettre au point un bon type objet ? Quelle méthode et quelle variable choisir ? etc. Or, il est très difficile de répondre à de telles questions. Seule l'expérience permet de progresser dans la conception objet, même si d'excellents ouvrages (comme par exemple [5]) permettent d'accélérer l'apprentissage. Le principal conseil d'apprentissage que nous pouvons donner est donc le suivant : pour apprendre à créer de bons types objet, il faut en programmer le plus possible !

Bien entendu, le programmeur doit quand même acquérir une certaine maîtrise technique, par exemple en appliquant les recommandations suivants :

- Il est bien sûr indispensable de commencer par comprendre le fonctionnement d'objets simples (comme ceux donnés en exemple), en particulier les mécanismes qui régissent les **variables et les méthodes d'instance** (notamment le `this`).
- Les **constructeurs** sont souvent mal compris. Il est donc important de consacrer du temps à leur apprentissage, par exemple en reproduisant leur mécanisme sur divers exemples et en l'illustrant par des dessins de la mémoire.
- Deux méthodes pratiques, **la conversion en chaîne de caractères** et la **comparaison**, obéissent à des mécanismes propres assez complexes. Comme ces méthodes sont souvent indispensables, il est important de retenir leur mode de fonctionnement.
- Le mécanisme de **contrôle d'accès** (`private` et `public`) est très simple du point de vue technique. Par contre, sa mise en œuvre est difficile car il faut toujours combattre la première impression fort naturelle : on se demande en effet quelle peut être l'utilité d'un mécanisme qui limite les possibilités du programmeur.
- Quand on maîtrise les éléments techniques, il faut passer à la **méthodologie objet** qui ne peut vraiment s'acquérir qu'en programmant. Les conseils donnés dans les sections 9.4 et 9.5 constituent un bon début et il est important de les appliquer scrupuleusement dans les premiers programmes développés.

CHAPITRE 10

Graphisme

Sommaire

Introduction

La réalisation d'une application munie d'une bonne *interface graphique* (fenêtres, menus, etc.) est une tâche assez délicate. Une interface moderne comporte des dizaines de menus, boites de dialogue, fenêtres, etc., ce qui aboutit à des centaines de lignes de code. De plus, la logique interne de l'application peut parfois être assez différente de la logique de l'interface. Si on songe par exemple à un logiciel permettant de jouer de la musique au format *mp3*, deux logiques s'affrontent. Pour le programmeur, la partie intéressante est le décodage du format *mp3*, qui fait intervenir un algorithme assez sophistiqué. L'utilisateur se moque complètement des subtilités du décodage et demande une interface ergonomique : sélection facile d'un morceau, enchaînements automatiques, etc.

Dans la pratique, il est donc indispensable en général de bien séparer la partie interface du reste du programme. Pour ce faire, l'emploi d'une approche objet est particulièrement appropriée. De plus, l'expérience prouve que seule une approche objet permet de limiter la taille du code nécessaire à l'écriture de l'interface graphique proprement dite. En `Java`, la question ne se pose même pas : pour produire des graphiques, même élémentaires, il faut impérativement utiliser des objets. Dans ce chapitre, nous ne prétendons pas donner un cours complet d'écriture d'interface graphique pour deux raisons : l'approche objet utilisée repose sur les concepts d'interface et d'héritage que nous n'avons pas abordés, et le domaine des interfaces graphiques est si riche qu'il demanderait quelques centaines de page de développement. Nous nous proposons donc de faire une rapide introduction au graphisme, qui permette l'écriture d'applications simples demandant une visualisation interactive des résultats. Pour compléter ce chapitre, nous encourageons le lecteur à se reporter aux documentations [12] et au tutoriel [3] fournis par `Sun`.

10.1 Principes fondamentaux

10.1.1 Deux systèmes graphiques

Le langage `Java` contient deux systèmes distincts destinés à la création d'interfaces graphiques. L'ancien système (développé à partir de la version `1.1` de `Java`) porte le nom de *Abstract Window Toolkit* (AWT), ce qui signifie (dans ce contexte) "bibliothèque abstraite de fenêtrage". Même si l'*AWT* est maintenant obsolète, il reste inclus dans `Java`. Il est cependant condamné à long terme.

Le nouveau système porte le nom de *Swing*. Il a été développé pour être compatible avec la version `1.1` de `Java` et est directement inclus dans le langage depuis la version `1.2`. *Swing* est une amélioration notable de *AWT*. Il est donc vivement conseillé de ne plus travailler qu'avec *Swing*, c'est pourquoi nous étudierons seulement ce système dans le présent ouvrage.

10.1.2 Composants graphiques *Swing*

Techniquement, *Swing* est un ensemble de classes, permettant de créer des objets. Chaque objet représente un **composant graphique**. Il existe de nombreux composants graphiques comme par exemple :

- `JFrame` : il s'agit de la fenêtre principale d'une application, celle qui possède un titre et une bordure, et éventuellement une barre de menus.
- `JButton` : il s'agit bien entendu d'un bouton, une zone rectangulaire sur laquelle on peut cliquer pour activer un élément d'un programme.
- `JPanel` : il s'agit d'une zone rectangulaire vide dans laquelle un programme peut dessiner. Un `JPanel` peut aussi contenir d'autres composants. Notons qu'un `JPanel` doit s'inclure dans un objet `JFrame` pour pouvoir être affiché. Nous utiliserons très souvent la classe `JPanel` pour proposer nos propres graphismes.
- `JMenuBar` : il s'agit de la barre de menus qui dans une `JFrame` contient les différents menus de l'application.
- `JMenu` : chaque objet `JMenu` représente un menu de l'application et s'insère dans une `JMenuBar`.
- `JMenuItem` : il s'agit simplement d'un texte à placer dans un menu, pour symboliser une option de ce dernier.

Il existe dans la pratique de nombreux autres composants graphiques possibles, comme par exemple les boites de dialogue, les ascenseurs, etc.

10.1.3 Création d'une fenêtre principale

Pour créer une application graphique `Java`, il est indispensable de proposer une fenêtre principale, l'idéal étant de passer par une `JFrame`. Commençons par l'exemple le plus élémentaire possible :

Exemple 10.1 :

On considère le programme suivant :

CreationFenetre

```
1  import javax.swing.*;
2  public class CreationFenetre {
3    public static void main(String[] args) {
4      JFrame fenêtre=new JFrame();
5      fenêtre.setVisible(true);
6    }
7  }
```

La ligne 1 correspond à l'utilisation d'une partie des classes qui constituent *Swing*. La ligne 4 crée l'objet `JFrame`, alors que la ligne 5 demande son affichage à l'écran. Le résultat est une petite fenêtre vide (avec un fond gris).

L'exemple précédent comporte plusieurs défauts :

1. la taille de la fenêtre n'est pas choisie par le programmeur.
2. le titre de la fenêtre n'est pas précisé par le programmeur.
3. quand on ferme la fenêtre, l'application ne se termine pas.

Il est relativement facile de remédier à ces problèmes en utilisant les méthodes adaptées, comme le montre l'exemple suivant :

Exemple 10.2 :

On complète l'exemple 10.1, ce qui donne le code suivant :

CreationFenetreComplete

```
1  import javax.swing.*;
2  public class CreationFenetreComplete {
3    public static void main(String[] args) {
4      JFrame fenêtre=new JFrame("Le titre");
5      fenêtre.setSize(250,300);
6      fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7      fenêtre.setVisible(true);
8    }
9  }
```

La fenêtre obtenue grâce au programme proposé est donnée par la figure 10.1.

Comme dans l'exemple 10.1, la ligne 4 crée la fenêtre. La seule nouveauté est le paramètre de type `String` qui permet de donner un titre à la fenêtre crée.

La ligne 5 fixe la taille de la fenêtre : le premier paramètre indique la largeur de la fenêtre, le second la hauteur.

La ligne 6 permet d'indiquer le comportement souhaité pour la fenêtre quand on la ferme. Notons que cette construction n'est pas utilisable quand on travaille avec une version de `Java` antérieure à la `1.3`. Quand on utilise la version `1.2`, il faut remplacer cette ligne par une construction assez complexe (dont nous ne détaillerons pas les mécanismes) présentée dans le programme suivant :

CreationFenetreCompleteOld

```
1  import javax.swing.*;
2  import java.awt.event.*;
3  public class CreationFenetreCompleteOld {
4    public static void main(String[] args) {
5      JFrame fenêtre=new JFrame("Le titre");
6      fenêtre.setSize(250,300);
7      fenêtre.addWindowListener(new WindowAdapter() {
8          public void windowClosing(WindowEvent e) {
9            System.exit(0);
10         }
11       });
12     fenêtre.setVisible(true);
13   }
14 }
```

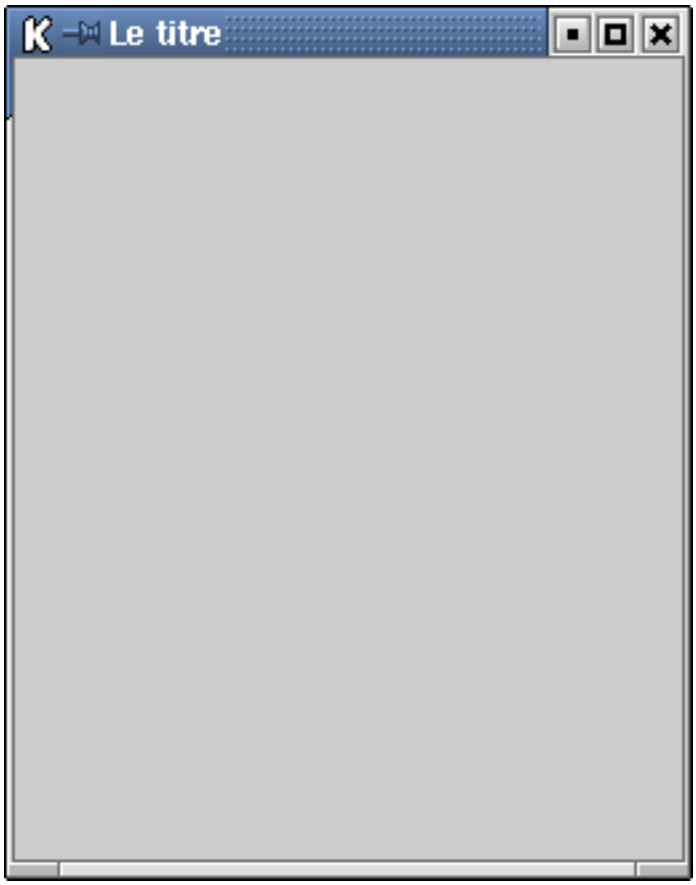

FIG. 10.1 – Affichage produit par la classe `CreationFenetreComplete`

Pour pouvoir utiliser cette construction (lignes 7 à 11), il faut impérativement ajouter un `import` (ligne 2).

REMARQUE

Il est en général inutile de préciser la taille de la fenêtre principale, car la méthode `pack` (que nous verrons à la section 10.1.5) permet d'adapter une fenêtre à son contenu.

Nous devons retenir les méthodes suivantes de la classe `JFrame` :

`JFrame(String title)`
: Ce constructeur permet de préciser le titre de la fenêtre crée.

`void setSize(int w,int h)`
: Permet de fixer les dimensions de la fenêtre (`w` désigne la largeur, `h` la hauteur).

`void setDefaultCloseOperation(int code)`
: Permet de choisir le comportement du programme quand on ferme la fenêtre. `code` est une valeur entière qu'on précise grâce à une constante de la classe `JFrame`. Par exemple `JFrame.EXIT_ON_CLOSE` correspond à un arrêt du programme en cas fermeture de la fenêtre principale.

`void setVisible(boolean t)`
: Indique si l'objet appelant est visible ou non. Dans la pratique, cette méthode permet de provoquer l'affichage de la fenêtre principale.

10.1.4 `JPanel` comme base pour le graphisme

Dans ce chapitre, nous nous intéressons avant tout à l'écriture de programme permettant des représentations graphiques. Pour ce faire, nous devons pouvoir dessiner, ce qui passe par la création d'un `JPanel` personnalisé. Le seul moyen de dessiner dans un `JPanel` est de fabriquer un objet graphique à partir de la classe `JPanel`, grâce à un mécanisme complexe appelé héritage qu'on peut considérer comme une technique de personnalisation d'une classe déjà existante. Pour ce faire, on se basera sur le modèle suivant :

DessinVide

```
1 import javax.swing.*;
2 import java.awt.*;
3 public class DessinVide extends JPanel {
4   public void paintComponent(Graphics g) {
5     super.paintComponent(g);
6     // on place ici les instructions de dessin
7   }
8 }
```

Nous verrons dans la suite du texte quelles sont les instructions qui permmettent d'obtenir un affichage à l'écran. Il suffit pour l'instant de savoir que ce sont les instructions de la méthode `paintComponent` qui prendront en charge l'affichage, **sans qu'il soit nécessaire d'appeler cette méthode**.

REMARQUE

Nous allons retrouver ici le mécanisme de `toString`, longuement étudié à la section 9.2.7. Comme pour cette méthode, il est **très important** de noter qu'il faut **impérativement** respecter scrupuleusement la définition de la méthode `paintComponent`, qui ne doit rien renvoyer, et surtout prendre un unique paramètre de type `Graphics`.

10.1.5 Comment afficher un `JPanel` personnalisé

Pour que la *magie* (l'appel automatique de `paintComponent`) décrite dans la section précédente puisse opérer, il faut bien sûr l'aider un peu. Pour ce faire, on doit compléter l'exemple 10.2, sur le modèle suivant :

Exemple 10.3 :

On propose le programme suivant :

AffichageDessinVide

```
1  import javax.swing.*;
2  import java.awt.*;
3  public class AffichageDessinVide {
4    public static void main(String[] args) {
5      JFrame fenêtre=new JFrame("Affichage de DessinVide");
6      fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7      DessinVide dessin=new DessinVide();
8      dessin.setPreferredSize(new Dimension(200,250));
9      fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
10     fenêtre.pack();
11     fenêtre.setVisible(true);
12   }
13 }
```

Analysons le programme ligne par ligne :

ligne 2 : cet `import` est indispensable pour pouvoir utiliser `BorderLayout`.

ligne 5 : création de la fenêtre principale, aucune différence avec l'exemple 10.2.

ligne 6 : on indique que le programme se termine quand on ferme la fenêtre principale.

ligne 7 : création de l'objet à afficher, c'est-à-dire de la version personnalisée du `JPanel`.

ligne 8 : grâce à la méthode `setPreferredSize`, on fixe la taille que le dessin aimerait avoir. Cette taille est une indication pour la fenêtre principale. Elle est donnée par un objet `Dimension`, dont le constructeur précise la largeur (premier paramètre) et la hauteur (second paramètre).

ligne 9 : insertion du dessin au centre de la fenêtre, ce qui est précisé par la constante de classe `BorderLayout.CENTER`.

ligne 10 : indique à la fenêtre de determiner sa taille en fonction des préférences des composants graphiques qu'elle contient.

ligne 11 : déclenche l'affichage de la fenêtre principale. **On ne peut pas afficher directement autre chose qu'une `JFrame`.**

Ceci est suffisant : il est inutile de placer du code Java après l'affichage de la fenêtre. C'est ici qu'intervient la *magie*. En fait, il n'y a rien de bien compliqué : comme on a ajouté le dessin à la fenêtre, montrer la fenêtre va nécessairement montrer le dessin, c'est-à-dire pour l'ordinateur appeler la méthode `paintComponent` de celui-ci.

Notons pour conclure cet exemple que comme le `DessinVide` ne comporte aucune instruction de dessin, l'affichage produit par le présent programme se limite à une fenêtre vide, comparable à celle obtenue grâce au programme de l'exemple 10.2.

L'exemple précédent illustre quelques nouvelles méthodes utiles de la classe `JFrame` :

`void pack()`

Demande à la fenêtre appelante d'ajuster sa taille en fonction de son contenu actuel. En général, cette méthode rend inutile le choix par le programmeur des dimensions de la fenêtre principale.

`Container getContentPane()`

Renvoie l'objet chargé de la gestion du contenu de l'objet appelant. En effet, un objet `JFrame` peut contenir d'autres composants graphiques (c'est même son but), mais la gestion de ces composants n'est pas réalisée directement par la `JFrame` elle-même, mais par un objet spécialisé.

La principale utilisation de `getContentPane` est celle donnée à la ligne 9 de l'exemple précédent : on enchaîne deux appels, afin d'insérer un composant graphique dans la `JFrame`. Pour ce faire, on utilise la méthode suivante de la classe `Container` :

`void add(Component comp,Object constraints)`

Insère un composant graphique (`comp`) dans le container appelant à la position indiqué par le paramètre `constraints`. Pour l'instant, nous nous contenterons de la position centrée, indiquée par `BorderLayout.CENTER`.

La ligne 8 de l'exemple précédent peut sembler étrange : on appelle une méthode d'instance de la classe `DessinVide`, alors que le code de celle-ci ne comporte pas la définition d'une telle méthode. En fait, c'est justement la définition *par extension* (techniquement par héritage) qui autorise ceci : toutes les méthodes de `JPanel` sont automatiquement utilisables dans `DessinVide`. Pour l'instant, nous avons donc découvert la méthode suivante de `JPanel` :

`void setPreferredSize(Dimension d)`

Indique les dimensions idéales de l'objet appelant. Ces dimensions sont obtenues grâce à un objet de type `Dimension`. On se contente en général de créer l'objet en question au moment de l'utilisation de la méthode, grâce au constructeur à deux paramètres illustré par l'appel suivant : `new Dimension(w,h)`. Le premier paramètre est la largeur de l'objet, alors que le second est sa hauteur.

10.2 Dessiner dans un `JPanel`

10.2.1 Paradigme

Le `JPanel` représente une zone rectangulaire de l'écran de l'ordinateur sur laquelle on peut dessiner. Pour ce faire, il faut tenir compte du modèle employé par celui-ci. Il est assez simple, mais un peu déroutant quand on est habitué à la représentation mathématique classique du plan.

La zone rectangulaire est constituée de points élémentaires, appelés *pixels*. Contrairement au point mathématique, un pixel possède une surface, c'est en fait un petit carré. Le `JPanel` est donc constitué d'une mosaïque de points.

Les points sont repérés par deux coordonnées **entières** (des `int`s) positives. L'origine du repère est le **coin supérieur gauche** de la toile : le pixel le plus en haut à gauche possède donc les coordonnées $(0, 0)$. L'axe des x (la première coordonnée) est orienté vers la droite (comme c'est la tradition en mathématiques). L'axe des y (la seconde coordonnée) est orienté **vers le bas** de l'écran. Pour obtenir les coordonnées d'un pixel, il suffit en fait de compter les nombres de colonnes et de lignes de pixels le séparant du pixel supérieur gauche (en comptant la colonne et la ligne de départ). Le tableau suivant donne les coordonnées des pixels dans une toile de dimension 3 sur 4 (3 lignes et 4 colonnes) :

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)

Remarquez la notation qui est l'opposée de celle utilisée traditionnellement pour les matrices !

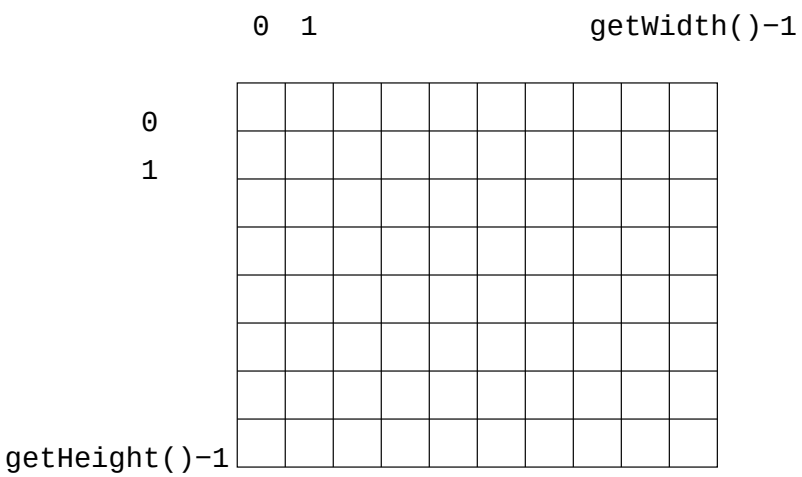

FIG. 10.2 – Dimensions du `JPanel`

10.2.2 Paramètres du `JPanel`

Dans une application fenêtrée, il est bien entendu possible de modifier la taille des fenêtres. Quand on modifie la fenêtre, on modifie aussi son contenu. Il faut donc pouvoir demander au programme la taille du `JPanel` pour pouvoir dessiner dans celui-ci.

Pour ce faire, on utilise le fait que l'objet graphique qu'on programme est obtenu par *héritage* à partir de `JPanel`. Or, cette classe propose deux méthodes très utiles :

`int getHeight()`

Renvoie la hauteur (en *pixels*) de l'objet appelant.

`int getWidth()`

Renvoie la largeur (en *pixels*) de l'objet appelant.

Comme toujours en `Java`, un `JPanel` à `n` lignes (`getHeight()`) et `p` colonnes (`getWidth()`) correspond à des pixels dont la coordonnée en x est comprise au sens large entre `0` et `p-1` et dont la coordonnée en y est comprise au sens large entre `0` et `n-1`, comme l'illustre la figure 10.2.

REMARQUE

Il est important de noter que la taille de la fenêtre est *dynamique*. En effet, l'utilisateur peut à tout moment changer les dimensions de la fenêtre principale, ce qui entraîne un appel automatique à `paintComponent`. On doit donc considérer que chaque appel de `paintComponent` se fait dans un `JPanel` avec de nouvelles dimensions.

10.2.3 Tracé de ligne

Pour dessiner dans la zone rectangulaire disponible, il faut donner un contenu à la méthode `paintComponent`. Pour réaliser le tracé effectif, on utilise l'objet `Graphics` transmis à la méthode `paintComponent`. Cet objet possède différentes méthodes de dessin, comme par exemple la méthode suivante :

`void drawLine(int x1,int y1,int x2,int y2)`

Cette méthode trace une ligne depuis le pixel de coordonnées (`x1`,`y1`) vers le pixel de coordonnées (`x2`,`y2`).

On peut en fait considérer l'objet `Graphics` comme le pinceau avec lequel on dessine.

Exemple 10.4 :

Voici par exemple un nouvel objet graphique, construit sur le modèle de `DessinVide`, qui dessine cette fois ci un cadre :

DessinCadre

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class DessinCadre extends JPanel {
4    public void paintComponent(Graphics g) {
5      super.paintComponent(g);
6      int h=getHeight();
7      int w=getWidth();
8      g.drawLine(1,1,w-2,1);
9      g.drawLine(w-2,1,w-2,h-2);
10     g.drawLine(w-2,h-2,1,h-2);
11     g.drawLine(1,h-2,1,1);
12   }
13 }
```

Pour effectivement utiliser cette toile, il suffit de remplacer la ligne 7 du programme `AffichageDessinVide` par :

```
DessinCadre dessin=new DessinCadre();
```

On obtient par exemple le programme pincipal suivant :

AffichageDessinCadre

```
1  import javax.swing.*;
2  import java.awt.*;
3  public class AffichageDessinCadre {
4    public static void main(String[] args) {
5      JFrame fenêtre=new JFrame("Affichage de DessinCadre");
6      fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7      DessinCadre dessin=new DessinCadre();
8      dessin.setPreferredSize(new Dimension(200,250));
9      fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
10     fenêtre.pack();
11     fenêtre.setVisible(true);
12   }
13 }
```

En essayant ce programme, vous remarquerez que le cadre occupe toujours la même position relative dans la fenêtre : ce comportement est obtenu grâce à l'utilisation des dimensions de la toile et est illustré par la figure 10.3.

Il faut bien comprendre ce qui se passe. La `JFrame` est affichée grâce à l'utilisation de la méthode `setVisible`. Comme le `DessinCadre` est inclus dans la fenêtre principale, il est aussi affiché, ce qui se traduit par l'appel automatique de la méthode `paintComponent`. Ensuite, quand l'utilisateur modifie la taille de la fenêtre, l'ordinateur appelle automatiquement `paintComponent`, pour obtenir le réaffichage du dessin, dans la fenêtre retaillée. Lors de ce nouvel appel de `paintComponent`, le dessin a changé de taille, et l'utilisation des méthodes `getHeight` et `getWidth` permet d'obtenir les nouvelles dimensions, et donc d'avoir un cadre de taille adapté.

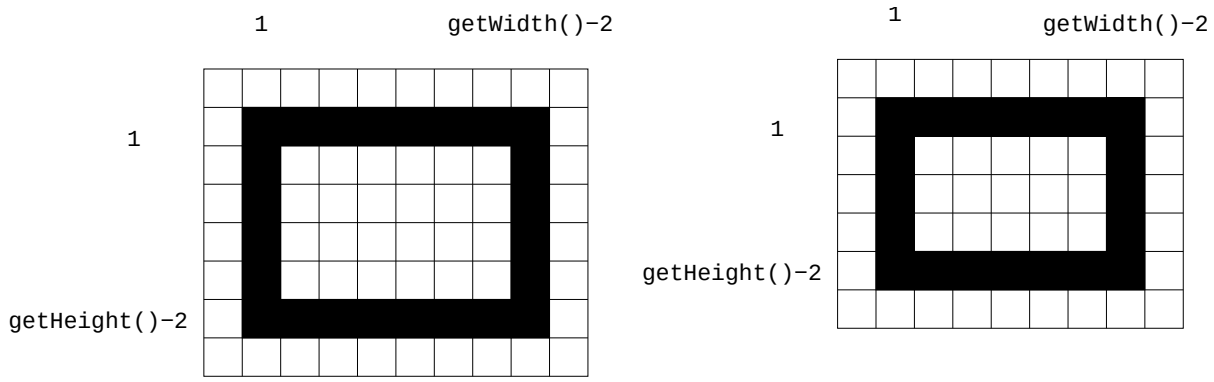

FIG. 10.3 – Exemple de tracé

REMARQUE

Si on oublie l'appel `super.paintComponent(g) ;` dans une méthode `paintComponent`, la compilation et l'exécution du programme résultat ne posent pas de problème particulier, au sens où l'ordinateur n'indique pas d'erreur. Par contre, l'affichage comportera parfois des *bugs*, essentiellement sous la forme de superposition de plusieurs dessins, par exemple plusieurs cadres dans l'exemple précédent.

10.2.4 Autres manipulations graphiques

L'essentiel des manipulations graphiques se fait par l'intermédiaire de la classe `Graphics`, c'est-à-dire grâce au paramètre `g` de la méthode `paint`. Les méthodes proposées par cette classe sont très nombreuses. Pour plus de renseignements, vous pouvez vous reporter à la documentation du `jdk` [12].

Voici quelques méthodes utiles :

`void setColor(Color c)`

Fixe la couleur que l'ordinateur va utiliser pour les prochains tracés. Les couleurs sont décrites par la classe `Color`. On peut définir une couleur en utilisant le constructeur de `Color` qui prend trois entiers compris entre 0 et 255. Chaque couleur est représentée informatiquement par un mélange de rouge, vert et bleu, dont la "quantité" est fixée par l'entier. Si on écrit par exemple :

```
Color c=new Color(15,160,12) ;
```

on définit une couleur composée de 15 unités de rouge, 160 unités de vert et 12 unités de bleu (sur un maximum de 255 pour chaque couleur). Il existe des couleurs pré-définies, qu'on utilise en écrivant par exemple `Color.red`[1], pour la couleur rouge. On peut utiliser de cette façon les couleurs suivantes : `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lighGray`, `magenta`, `orange`, `pink`, `red`, `white` et `yellow`.

`void drawOval(int x,int y,int width,int height)`

Trace une ellipse tenant exactement dans le rectangle décrit par les 4 paramètres numériques de la méthode : `x` et `y` sont les coordonnées du coin supérieur gauche du rectangle, `x+width` et `y+height` sont les coordonnées du coin inférieur droit du rectangle.

`void fillOval(int x,int y,int width,int height)`

Même chose que la méthode précédente, mais en remplissant l'ellipse avec la couleur du tracé.

[1] Il s'agit donc de constantes de la classe `Color`, qui n'utilisent malheureusement pas les conventions pour les noms de constantes !

`void drawString(String s,int x,int y)`

Affiche le texte contenu dans `s` en partant du point de coordonnées (`x`,`y`).

10.3 Écrire un `JPanel` paramètrable

10.3.1 Motivation

Le principal défaut de l'exemple 10.4 proposé dans la section précédente est qu'il est complètement figé. Si on veut par exemple changer la couleur, l'épaisseur, etc. du cadre, on doit modifier la classe `DessinCadre` puis la recompiler, ce qui n'est pas très satisfaisant. Nous souhaitons au contraire pouvoir préciser à l'exécution du programme les paramètres du tracé, en utilisant par exemple une saisie grâce aux méthodes de la classe `Console`. Pour ce faire, il faut que les paramètres du tracé soit transmis à la méthode `paintComponent`, ce qui l'objet de la section suivante.

10.3.2 Un `JPanel` est un objet

Notons tout d'abord que la transmission de paramètres n'est pas simple. En effet, comme nous l'avons déjà précisé, la méthode `paintComponent` **ne peut en aucun cas recevoir de paramètres additionnels**. Son seul paramètre est une référence vers un objet `Graphics`. Il faut donc utiliser un autre mécanisme : l'objet appelant.

Notons en effet que la méthode `paintComponent` ne comporte pas le mot clé `static`. C'est donc une méthode d'instance. De ce fait, elle contient un paramètre par défaut, `this`, qui fait référence à l'objet appelant, c'est-à-dire à l'objet graphique. Il suffit donc de placer dans des variables d'instance de cet objet les paramètres du tracé pour que ceux-ci soient accessibles par la méthode `paintComponent`. Grâce au constructeur de l'objet, on peut ensuite fixer les valeurs des paramètres.

Exemple 10.5 :

Voici une nouvelle version de `DessinCadre` qui propose des paramètres et un constructeur :

DessinCadreParam

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class DessinCadreParam extends JPanel {
4    private int distance;
5    private int épaisseur;
6    public DessinCadreParam(int d,int e) {
7      distance=d;
8      épaisseur=e;
9    }
10   public void dessineCadre(Graphics g,int d) {
11     int h=getHeight();
12     int w=getWidth();
13     g.drawLine(d,d,w-d-1,d);
14     g.drawLine(w-d-1,d,w-d-1,h-d-1);
15     g.drawLine(w-d-1,h-d-1,d,h-d-1);
16     g.drawLine(d,h-d-1,d,d);
17   }
18   public void paintComponent(Graphics g) {
19     super.paintComponent(g);
20     for(int i=distance;i<distance+épaisseur;i++)
21       dessineCadre(g,i);
```

```
22    }
23  }
```

Voici quelques explications sur cette nouvelle version :
- la variable d'instance `distance` représente la distance entre le bord de la fenêtre et le bord du cadre qu'on va dessiner. La variable `épaisseur` représente l'épaisseur du cadre à dessiner.
- la méthode `dessineCadre` reprend à peu près le code de la méthode `paintComponent` de `DessinCadre` : elle dessine un cadre d'un pixel d'épaisseur, situé à `d` pixels du bord de la fenêtre.
- la méthode `paintComponent`, qui réalise le dessin proprement dit, est basé sur des appels successifs à `dessineCadre`.

Bien entendu, l'utilisation de la nouvelle classe demande une nouvelle classe principale :

AffichageDessinCadreParam

```
1  import javax.swing.*;
2  import java.awt.*;
3  import dauphine.util.*;
4  public class AffichageDessinCadreParam {
5    public static void main(String[] args) {
6      Console.start();
7      System.out.print("Distance au bord : ");
8      int d=Console.readInt();
9      System.out.print("Epaisseur : ");
10     int e=Console.readInt();
11     JFrame fenêtre=new JFrame("Affichage de DessinCadreParam");
12     fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13     DessinCadreParam dessin=new DessinCadreParam(d,e);
14     dessin.setPreferredSize(new Dimension(200,250));
15     fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
16     fenêtre.pack();
17     fenêtre.setVisible(true);
18   }
19 }
```

La nouvelle classe est une version plus complète de `AffichageDessinCadre` dont les premières lignes permettent à l'utilisateur de choisir la distance du cadre au bord de la fenêtre ainsi que l'épaisseur du cadre. Quand l'utilisateur saisit par exemple les valeurs 5 pour la distance au bord et 10 pour l'épaisseur, le programme affiche la fenêtre indiquée par la figure 10.4.

10.3.3 Cas d'un objet modifiable

Dans l'exemple précédent, `DessinCadreParam` est un objet *immuable* : après sa création, il est impossible de modifier ses variables d'instance. Dans la pratique, il n'est bien entendu pas obligatoire de proposer un `JPanel` immuable. Il est même parfois plus pratique d'avoir un `JPanel` modifiable. Pour l'exemple qui nous occupe, on peut proposer des méthodes permettant de modifier l'épaisseur et la distance au bord du cadre. Cela ne pose pas vraiment de problème, mais il faut tenir compte d'une subtilité. Quand on change un paramètre du dessin, celui-ci doit impérativement être réaffiché. Or, il n'est pas possible d'appeler directement la méthode `paintComponent` (il faudrait pour cela fabriquer un objet `Graphics`, ce qui n'est pas possible dans les conditions étudiées). On doit donc passer par un mécanisme spécifique, la méthode `repaint`. L'appel de cette méthode produit le réaffichage de l'objet appelant, l'ordinateur se chargeant d'appeler correctement

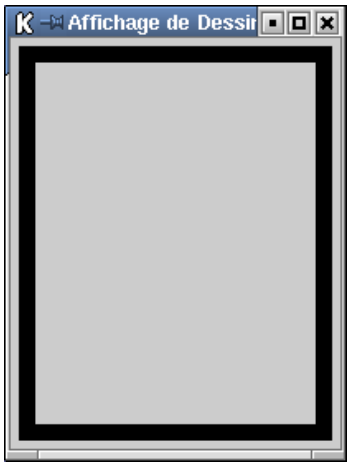

FIG. 10.4 – Affichage produit par la classe `AffichageDessinCadreParam`

la méthode `paintComponent`. La méthode est *héritée* par tous les composants graphiques. Dans notre cas, nous programmons une classe qui hérite de `JPanel` et nous pouvons appeler directement `repaint` dans n'importe quelle méthode d'instance.

Exemple 10.6 :

Reprenons l'exemple 10.5 en rendant les objets de type `DessinCadreParam` modifiables. Pour obtenir la classe `DessinCadreParamMod`, on ajoute à la classe `DessinCadreParam` les méthodes suivantes :

DessinCadreParamMod

```
1  public void setDistance(int d) {
2    distance=d;
3    repaint();
4  }
5  public void setEpaisseur(int e) {
6    épaisseur=e;
7    repaint();
8  }
```

Les deux méthodes sont très simples et on remarque l'appel à `repaint` permettant le réaffichage. Une application simple de la nouvelle `Toile` est proposée dans la classe suivante :

AffichageDessinMod

```
1  import javax.swing.*;
2  import java.awt.*;
3  import dauphine.util.*;
4  public class AffichageDessinMod {
5    public static void main(String[] args) {
6      Console.start();
7      System.out.print("Distance au bord : ");
8      int d=Console.readInt();
9      System.out.print("Epaisseur : ");
10     int e=Console.readInt();
11     JFrame fenêtre=new JFrame("Affichage de DessinCadreParamMod");
12     fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
13      DessinCadreParamMod dessin=new DessinCadreParamMod(d,e);
14      dessin.setPreferredSize(new Dimension(200,250));
15      fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
16      fenêtre.pack();
17      fenêtre.setVisible(true);
18      do {
19        System.out.print("Epaisseur : ");
20        e=Console.readInt();
21        if(e>0) {
22          dessin.setEpaisseur(e);
23        }
24      } while(e>0);
25    }
26  }
```

Le début de cette classe est identique à celui de la classe `AffichageDessinCadreParam` étudiée dans l'exemple 10.5. Par contre, les dernières lignes sont nouvelles (lignes 18 à 24). Leur principe est simple : une boucle demande à l'utilisateur de saisir une nouvelle épaisseur, puis modifie l'objet auquel `dessin` fait référence (une `DessinCadreParamMod`). Comme la méthode `setEpaisseur` de la classe en question contient un appel à `repaint`, chaque modification de l'épaisseur du cadre va se traduire par la modification de l'affichage : le cadre aura la nouvelle épaisseur choisie.

Le seul point délicat de cet exemple réside dans l'utilisation (cachée) d'un effet de bord. Il faut bien comprendre en effet que l'objet auquel `dessin` fait référence est celui que la `JFrame` utilise pour produire l'affichage. Il y a en fait un unique objet de type `DessinCadreParamMod` et c'est pourquoi une modification de cet objet provoque une modification de l'affichage graphique.

10.4 Interaction avec l'utilisateur

10.4.1 Introduction

Dans la section 10.3.3, nous avons vu comment utiliser un `JPanel` modifiable afin de proposer des éléments d'interaction avec l'utilisateur du programme. La classe `AffichageDessinMod` (exemple 10.6) permet par exemple de modifier l'épaisseur du cadre affiché en tapant au clavier une nouvelle épaisseur. Ce type d'interaction est pratique, mais reste limité par rapport à ce que proposent des programmes évolués. Le but de cette section est de présenter des techniques permettant de proposer une interactivité plus évoluée que ce que nous avons déjà étudié dans les sections précédentes.

10.4.2 Exemple de gestion du clavier

Il est possible de gérer relativement simplement le clavier de l'ordinateur, afin qu'une simple pression sur une touche provoque une modification de l'affichage proposé. Dans l'exemple du cadre, nous allons faire en sorte qu'une pression sur la touche `+` provoque une augmentation de l'épaisseur du cadre, alors qu'une pression sur la touche `-` provoquera au contraire une diminution de l'épaisseur.

Avant de proposer un gestionnaire de clavier, nous complétons la classe `DessinCadreMod` en ajoutant la méthode suivante :

DessinCadreComplet

```
1  public int getEpaisseur() {
```

```
2   return épaisseur;
3 }
```

Pour réaliser noter but (modification de l'épaisseur du cadre), nous proposons la classe suivante :

ControleCadre

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class ControleCadre extends KeyAdapter {
4    DessinCadreComplet cadre;
5    public ControleCadre(DessinCadreComplet t) {
6      cadre=t;
7    }
8    public void keyPressed(KeyEvent e) {
9      char c=e.getKeyChar();
10     if(c=='+') {
11       cadre.setEpaisseur(cadre.getEpaisseur()+1);
12     } else if(c=='-') {
13       int ep=cadre.getEpaisseur();
14       if(ep>0) {
15         cadre.setEpaisseur(ep-1);
16       }
17     }
18   }
19 }
```

Des explications s'imposent :

- nous remarquons la présence du mot-clé `extends` qui indique que la classe `ControleCadre` est définie par extension à partir de la classe `KeyAdapter`. **Pour pouvoir réagir à la pression des touches du clavier, il faut obligatoirement définir une classe par extension à partir de** `KeyAdapter` ;
- la méthode `keyPressed` reçoit un objet de type `KeyEvent`. Par un mécanisme que nous décrirons brièvement dans la suite de cette section (puis en détail dans la section suivante), cette méthode est automatiquement appelée par l'ordinateur quand l'utilisateur presse une touche du clavier (d'où le nom *key pressed*). De plus (un peu comme pour la méthode `paint`), l'ordinateur fournit à la méthode un paramètre adapté, qui décrit ici la touche qui a été enfoncée. Plus précisément, comme l'indique la suite de la méthode, on peut obtenir le caractère inscrit sur la touche appuyée en appelant la méthode d'instance `getKeyChar` de la classe `KeyEvent`. La suite de la méthode `keyPressed` est assez claire : si on appuie sur `+`, l'épaisseur du cadre augmente (lignes 10 et 11), alors qu'elle diminue (si possible) quand on appuie sur la touche `-`. Notons que **pour pouvoir réagir à la pression des touches du clavier, il faut obligatoirement définir une méthode** `KeyPressed` **comme nous venons de le faire**.

Bien entendu, la simple définition de la classe `ControleCadre` n'est pas suffisante. Il faut indiquer à l'ordinateur qu'il doit utiliser cette classe pour réagir à la pression d'une touche. Pour ce faire, on utilise une nouvelle classe principale :

AffichageDessinControle

```
1  import javax.swing.*;
2  import java.awt.*;
3  import dauphine.util.*;
4  public class AffichageDessinControle {
```

```
5   public static void main(String[] args) {
6     Console.start();
7     System.out.print("Distance au bord : ");
8     int d=Console.readInt();
9     System.out.print("Epaisseur : ");
10    int e=Console.readInt();
11    JFrame fenêtre=new JFrame("Affichage de DessinCadreComplet");
12    fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13    DessinCadreComplet dessin=new DessinCadreComplet(d,e);
14    dessin.setPreferredSize(new Dimension(200,250));
15    fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
16    fenêtre.pack();
17    fenêtre.setVisible(true);
18    ControleCadre contrôleur=new ControleCadre(dessin);
19    dessin.addKeyListener(contrôleur);
20    dessin.requestFocus();
21  }
22 }
```

On voit que cette classe comporte trois lignes utilisant des constructions nouvelles :

ligne 18 : on crée ici une instance de la classe `ControleCadre`. Le constructeur prend comme paramètre `dessin` qui désigne l'objet de type `DessinCadreComplet` qui dessine un cadre. De ce fait, si la méthode `keyPressed` de la classe `ControleCadre` est appelée, c'est bien l'objet contenu dans le cadre qui sera modifié. Comme dans l'exemple de la classe `AffichageDessinMod` (exemple 10.6), c'est un effet de bord qui assure le fonctionnement général du programme.

ligne 19 : plus mystérieuse, cette ligne indique à l'objet de dessin, qu'il doit utiliser comme gestionnaire du clavier la classe `ControleCadre`.

ligne 20 : cette ligne indique que l'objet graphique (de type `DessinCadreComplet`) demande que les pressions des touches lui soient transmisses. Si on oublie cette ligne, l'application ne réalise rien de bien intéressant car les pressions des touches sont indiquées à d'autres composants graphiques. Notons que seule l'interaction basée sur le clavier demande l'utilisation de cette méthode.

Quand on lance l'application ainsi construite, l'ordinateur commence par demander les paramètres du dessin, puis affiche normalement un cadre. Quand on place le pointeur de la souris dans la fenêtre et qu'on appuie sur la touche `+`, l'épaisseur du cadre augmente. Si on appuie sur la touche `-`, l'épaisseur diminue, ce qui est bien le comportement voulu.

10.4.3 Le mécanisme de gestion des évènements

L'exemple proposé dans la section précédente reste assez mystérieux. Voyons plus en détail le mécanisme utilisé par `Java` pour gérer les évènements.

Les évènements

Dans le contexte qui nous intéresse (les interfaces graphiques), un **évènement** est la traduction informatique d'une action de l'utilisateur du programme, ce qui inclut : un mouvement de la souris, la pression d'un des boutons de la souris, la pression d'une touche du clavier, la fermeture de la fenêtre du programme, etc. En `Java`, l'ordinateur représente les évènements par des objets. Nous nous intéressons dans un premier temps aux évènements correspondant au clavier de l'ordinateur

qui sont représentés par la classe `KeyEvent`. Cette classe définit diverses méthodes, dont seules deux nous intéressent ici :

`char getKeyChar()`

Cette méthode renvoie sous forme d'un `char` la touche qui a été enfoncée.

`int getKeyCode()`

Cette méthode renvoie sous forme d'un code spécifique à `Java` la touche qui a été enfoncée.

La différence principale entre les deux méthodes est que la deuxième permet d'étudier les touches spéciales (comme par exemple les touches de déplacement du curseur) qui ne correspondent pas à un caractère.

Quand l'utilisateur appuie sur une touche, l'ordinateur fabrique donc un objet de type `KeyEvent`, contenant la description de la touche enfoncée.

La réaction aux évènements

Comme nous l'avons dit dans la section 10.1.2, une interface graphique est constituée de *composants*. Quand l'utilisateur réalise une action (qui se traduit donc par un évènement), l'ordinateur est capable de déterminer le composant *destinataire* de l'évènement. Pour traiter l'évènement, l'ordinateur le transmet à ce composant destinataire. Le comportement par défaut des composants est de ne rien faire quand ils reçoivent un évènement. Pour modifier ce comportement, il faut enregistrer auprès du composant choisi un objet qui sera utilisé pour traiter l'évènement. Dans le cas qui nous intéresse (la gestion du clavier) ceci s'effectue grâce à la méthode `addKeyListener`, définie directement au niveau de la classe `Component` :

`void addKeyListener(KeyListener l)`

Ajoute au composant appelant l'objet `l` pour la gestion des évènements clavier.

On peut traduire `addKeyListener` par "ajouter un guetteur de clavier". Pour simplifier, on peut dire que cette méthode enregistre l'objet transmis en paramètre, de manière à ce que l'ordinateur l'utilise pour réagir aux évènements clavier qui ont pour destinataire le composant appelant la méthode.

Un guetteur de clavier

Il nous faut maintenant comprendre comment programmer un *guetteur de clavier*. Donner une explication complète n'est pas possible dans le cadre de ce cours. Pour simplifier, on peut dire que la programmation d'un tel objet peut se faire par extension à partir de la classe `KeyAdapter`. La forme générale d'un guetteur de clavier sera la suivante (il faut respecter **scrupuleusement** les en-têtes des méthodes) :

GuetteurClavier

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class GuetteurClavier extends KeyAdapter {
4    // variable(s) d'instance
5    // constructeur(s)
6    public void keyPressed(KeyEvent e) {
7      // touche enfoncée
8    }
9    public void keyReleased(KeyEvent e) {
10     // touche relachée
11   }
```

```
12    public void keyTyped(KeyEvent e) {
13      // touche appuyée puis relachée
14    }
15  }
```

Pour programmer un guetteur de clavier, on se basera donc sur le squelette proposé, **en ne définissant que les méthodes utiles** :

`void keyPressed(KeyEvent e)`
: Méthode appelée par l'ordinateur pour gérer un évènement clavier quand une touche est enfoncée. L'ordinateur n'attend pas que la touche soit relâchée pour appeler cette méthode. Cette méthode comporte l'avantage d'être appelée même si la touche enfoncée ne correspond pas directement à un caractère (par exemple la touche de majuscule).

`void keyReleased(KeyEvent e)`
: Méthode appelée par l'ordinateur pour gérer un évènement clavier quand une touche est relâchée. Comme `keyPressed`, la méthode est appelée même pour les touches spéciales.

`void keyTyped(KeyEvent e)`
: Méthode appelée par l'ordinateur pour gérer un évènement clavier quand une touche est appuyée puis relâchée (un seul appel). Cette méthode n'est appelée que pour les caractères, par pour les touches de fonctions. Il est possible que la pression d'une combinaison de touche ne provoque qu'un seul appel à cette méthode : par exemple majuscule suivi de `a` provoque un seul appel à `keyTyped`, avec comme paramètre un `KeyEvent` correspond au caractère `A`.

Exemple élémentaire

Voici un exemple simple de guetteur, qui se contente d'afficher les évènements reçus, afin de bien illustrer le comportement de l'ordinateur :

GuetteurDisplay

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class GuetteurDisplay extends KeyAdapter {
4    public void displayEvent(String where,KeyEvent e) {
5      System.out.println(where+" char : "+e.getKeyChar());
6      System.out.println(where+" code : "+e.getKeyCode());
7    }
8    public void keyPressed(KeyEvent e) {
9      displayEvent("pressed",e);
10   }
11   public void keyReleased(KeyEvent e) {
12     displayEvent("released",e);
13   }
14   public void keyTyped(KeyEvent e) {
15     displayEvent("typed",e);
16   }
17 }
```

Pour utiliser ce guetteur, il faut bien sûr l'enregistrer auprès d'un composant. Pour obtenir l'exemple le plus simple possible, nous enregistrons le guetteur auprès du dessin vide de la section 10.1.4, comme le montre le programme suivant :

DemoGuetteurDisplay

```
1  import javax.swing.*;
2  import java.awt.*;
3  public class DemoGuetteurDisplay {
4    public static void main(String[] args) {
5      JFrame fenêtre=new JFrame("Démonstration de GuetteurDisplay");
6      fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7      DessinVide dessin=new DessinVide();
8      dessin.setPreferredSize(new Dimension(200,250));
9      fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
10     fenêtre.pack();
11     fenêtre.setVisible(true);
12     // création du guetteur
13     GuetteurDisplay g=new GuetteurDisplay();
14     // enregistrement
15     dessin.addKeyListener(g);
16     // on souhaite recevoir les évènements clavier
17     dessin.requestFocus();
18   }
19 }
```

Quand on lance le programme, une fenêtre vide s'affiche à l'écran (voir la figure 10.1). Quand on place le pointeur de la souris dans la fenêtre puis qu'on appuie sur les touches du clavier, des messages s'affichent dans la fenêtre de lancement du programme (pas dans la fenêtre correspondant à au `DessinVide`, bien entendu). Si par exemple on appuie puis on relâche la touche `a`, on obtient :

```
pressed char : a
pressed code : 65
typed char : a
typed code : 0
released char : a
released code : 65
```

Si on appuie puis relâche la touche majuscule, on obtient :

```
pressed char : ?
pressed code : 16
released char : ?
released code : 16
```

ce qui montre bien que la méthode `keyTyped` n'est pas appelée pour les touches spéciales. Si on appuie sur majuscule, puis sur `a` (en maintenant majuscule enfoncée), puis qu'on relâche `a`, puis enfin qu'on relâche la touche `majuscule`, on obtient l'affichage suivant :

```
pressed char : ?
pressed code : 16
pressed char : A
pressed code : 65
typed char : A
typed code : 0
released char : A
released code : 65
```

On voit que l'utilisation de `keyTyped` est plus simple pour l'ordinateur. En effet, dans le cas d'une combinaison de touches, on obtient un seul évènement. Notons que la ligne 17 (`dessin.requestFocus() ;`) est absolument indispensable au bon fonctionnement du programme. En son absence, l'objet `DessinVide` ne reçoit pas les évènements clavier, et les méthodes correspondantes de son guetteur de clavier ne sont jamais appelées.

Pour obtenir un comportement plus complexe, il faut bien entendu adapter le squelette `GuetteurClavier` en ajoutant par exemple des variables d'instance, un constructeur, etc. La classe `ControleCadre` de la section précédente donne un exemple plus réaliste de contrôle au clavier d'une application.

REMARQUE

Les codes utilisés par `Java` pour représenter les touches spéciales (par exemple 16 pour la touche majuscule d'après l'exemple précédent) ne doivent pas être devinés ou utilisés comme des entiers. La bonne façon d'utiliser `getKeyCode` passe par l'emploi de constantes de la classe `KeyEvent`. De même que la classe `Math` définit les constantes `Math.PI` et `Math.E`, la classe `KeyEvent` définit par exemple la constante `KeyEvent.VK_SHIFT` pour désigner la touche *shift* (c'est-à-dire majuscule), `KeyEvent.VK_UP` pour la touche "flèche vers le haut", etc. On se reportera à la documentation du kit de développement `Sun` [12] pour la liste des codes des touches spéciales.

10.4.4 Gestion de la souris

Évènement souris

Comme nous l'avons indiqué dans la section précédente, la notion d'évènement recouvre les actions effectuées à la souris par l'utilisateur. Pour représenter ces actions, l'ordinateur utilise la classe `MouseEvent` qui comporte (entre autres) les méthodes suivantes :

`int getX()`
: Renvoie la coordonnée horizontale de la souris au moment de l'évènement.

`int getY()`
: Renvoie la coordonnée verticale de la souris au moment de l'évènement.

Pour déterminer le bouton concerné par la pression (bouton de gauche, du centre ou de droite), il faut utiliser la classe `SwingUtilities` qui définit (entre autre) les méthodes **de classe** suivantes :

`boolean isLeftMouseButton(MouseEvent me)`
: Renvoie `true` si et seulement si `me` correspond à une pression du bouton gauche de la souris.

`boolean isMiddleMouseButton(MouseEvent me)`
: Renvoie `true` si et seulement si `me` correspond à une pression du bouton central de la souris.

`boolean isRightMouseButton(MouseEvent me)`
: Renvoie `true` si et seulement si `me` correspond à une pression du bouton droit de la souris.

Guetteurs de souris

Comme pour le clavier, les composants de `Java` ne réagissent pas naturellement à la souris. Pour programmer une réaction, il faut mettre en place *guetteur de souris*. Pour être précis, `Java` définit deux guetteurs de souris : le guetteur définit par extension à partir de `MouseAdapter` s'occupe des évènements ponctuels, c'est-à-dire de la réaction à la pression sur un bouton ou à l'entrée du pointeur de la souris dans un composant ; le guetteur définit à partir de `MouseMotionAdapter` s'occupe de son côté des évènements correspondant aux mouvements du pointeur de la souris dans un composant donné.

Voici la forme générale d'un guetteur de clic souris :

GuetteurSourisClic

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class GuetteurSourisClic extends MouseAdapter {
4    // variable(s) d'instance
5    // constructeur(s)
6    public void mouseClicked(MouseEvent e) {
7      // clic
8    }
9    public void mouseEntered(MouseEvent e) {
10     // entrée
11   }
12   public void mouseExited(MouseEvent e) {
13     // sortie
14   }
15   public void mousePressed(MouseEvent e) {
16     // bouton enfoncé
17   }
18   public void mouseReleased(MouseEvent e) {
19     // bouton relaché
20   }
21 }
```

Pour programmer un guetteur de clic souris, on se basera donc sur le squelette proposé, en ne définissant que les méthodes utiles :

`void mouseClicked(MouseEvent e)`
: Méthode appelée en cas de clic, c'est-à-dire de bouton enfoncé puis relâché (sans **aucun** mouvement de la souris entre les deux actions).

`void mouseEntered(MouseEvent e)`
: Méthode appelée quand le pointeur de la souris entre dans le composant destinataire de l'évènement.

`void mouseExited(MouseEvent e)`
: Méthode appelée quand le pointeur de la souris sort du composant destinataire de l'évènement.

`void mousePressed(MouseEvent e)`
: Méthode appelée quand un bouton de la souris est enfoncé.

`void mouse(MouseEvent e)`
: Méthode appelée quand un bouton de la souris préalable enfoncé est relâché.

Voici la forme générale d'un guetteur de mouvement de la souris :

GuetteurSourisMove

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class GuetteurSourisMove extends MouseMotionAdapter {
4    // variable(s) d'instance
5    // constructeur(s)
6    public void mouseDragged(MouseEvent e) {
```

```
7     // mouvement de la souris avec un bouton enfoncé
8   }
9   public void mouseMoved(MouseEvent e) {
10    // mouvement de la souris sans bouton enfoncé
11  }
12 }
```

Pour programmer un guetteur de mouvement de la souris, on se basera donc sur le squelette proposé, en ne définissant que les méthodes utiles :

`void mouseDragged(MouseEvent e)`
: Méthode appelée quand la souris est déplacée avec un bouton maintenu enfoncé.

`void mouseMoved(MouseEvent e)`
: Méthode appelée quand la souris est déplacée sans bouton enfoncé.

Pour enregistrer des guetteurs de souris, on utilise deux méthodes définies directement au niveau de la classe `Component` :

`void addMouseListener(MouseListener l)`
: Ajoute au composant appelant l'objet `l` pour la gestion des évènements clic de souris.

`void addMouseMotionListener(MouseMotionListener l)`
: Ajoute au composant appelant l'objet `l` pour la gestion des évènements mouvement de souris.

Exemples élémentaires

Commençons par un exemple simple de guetteur de clic de souris. Comme pour le guetteur de clavier, on se contente ici d'afficher un message pour chaque évènement :

GuetteurSourisClicDisplay

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  public class GuetteurSourisClicDisplay extends MouseAdapter {
5    public void displayEvent(String where,MouseEvent e) {
6      System.out.print(where+" : ("+e.getX()+","+e.getY()+")");
7      if(SwingUtilities.isLeftMouseButton(e)) {
8        System.out.println(" bouton de gauche");
9      } else if(SwingUtilities.isMiddleMouseButton(e)) {
10       System.out.println(" bouton du milieu");
11     } else if(SwingUtilities.isRightMouseButton(e)) {
12       System.out.println(" bouton de droite");
13     } else {
14       System.out.println();
15     }
16   }
17   public void mouseClicked(MouseEvent e) {
18     displayEvent("clicked",e);
19   }
20   public void mouseEntered(MouseEvent e) {
21     displayEvent("entered",e);
22   }
23   public void mouseExited(MouseEvent e) {
```

```
24      displayEvent("exited",e);
25    }
26    public void mousePressed(MouseEvent e) {
27      displayEvent("pressed",e);
28    }
29    public void mouseReleased(MouseEvent e) {
30      displayEvent("released",e);
31    }
32  }
```

On utilise ce guetteur en l'enregistrant auprès d'un composant, comme dans l'exemple suivant :

DemoGuetteurMCD

```
1   import javax.swing.*;
2   import java.awt.*;
3   public class DemoGuetteurMCD {
4     public static void main(String[] args) {
5       JFrame fenêtre=new JFrame("Démonstration de GuetteurSourisClicDisplay");
6       fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7       DessinVide dessin=new DessinVide();
8       dessin.setPreferredSize(new Dimension(200,250));
9       fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
10      fenêtre.pack();
11      fenêtre.setVisible(true);
12      // création du guetteur
13      GuetteurSourisClicDisplay g=new GuetteurSourisClicDisplay();
14      // enregistrement
15      dessin.addMouseListener(g);
16    }
17  }
```

Quand on lance ce programme, une fenêtre vide s'affiche. Quand on place le pointeur de la souris dans la fenêtre, un évènement `mouseEntered` se produit. Dans notre programme, cela se traduit par un affichage et on obtient par exemple :

```
entered : (247,242)
```

En laissant la souris dans la fenêtre, et en cliquant sur le bouton, on obtient l'affichage suivant :

```
pressed : (137,107) bouton de gauche
released : (137,107) bouton de gauche
clicked : (137,107) bouton de gauche
```

Si on place ensuite le pointeur de la souris en dehors de la fenêtre, on obtient un évènement de sortie (`mouseExited`), ce qui provoque par exemple l'affichage suivant :

```
exited : (328,248)
```

Notons qu'un évènement `mouseClicked` ne se produit que si on appuie et relâche le bouton de la souris *sans bouger celle-ci*. Dans le cas contraire, on obtient par exemple :

```
pressed : (64,129) bouton du milieu
released : (92,139) bouton du milieu
```

Pour le guetteur de mouvement de souris, on propose une version très proche :

GuetteurSourisMoveDisplay

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  public class GuetteurSourisMoveDisplay extends MouseMotionAdapter {
5    public void displayEvent(String where,MouseEvent e) {
6      System.out.print(where+" : ("+e.getX()+","+e.getY()+")");
7      if(SwingUtilities.isLeftMouseButton(e)) {
8        System.out.println(" bouton de gauche");
9      } else if(SwingUtilities.isMiddleMouseButton(e)) {
10       System.out.println(" bouton du milieu");
11     } else if(SwingUtilities.isRightMouseButton(e)) {
12       System.out.println(" bouton de droite");
13     } else {
14       System.out.println();
15     }
16   }
17   public void mouseDragged(MouseEvent e) {
18     displayEvent("dragged",e);
19   }
20   public void mouseMoved(MouseEvent e) {
21     displayEvent("moved",e);
22   }
23 }
```

Encore une fois, il faut enregister le guetteur auprès du composant :

DemoGuetteurMMD

```
1  import javax.swing.*;
2  import java.awt.*;
3  public class DemoGuetteurMMD {
4    public static void main(String[] args) {
5      JFrame fenêtre=new JFrame("Démonstration de GuetteurSourisMoveDisplay");
6      fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7      DessinVide dessin=new DessinVide();
8      dessin.setPreferredSize(new Dimension(200,250));
9      fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
10     fenêtre.pack();
11     fenêtre.setVisible(true);
12     // création du guetteur
13     GuetteurSourisMoveDisplay g=new GuetteurSourisMoveDisplay();
14     // enregistrement
15     dessin.addMouseMotionListener(g);
16   }
17 }
```

Comme les autres, ce programme affiche une fenêtre vide. Dès qu'on place le pointeur de la souris dans la fenêtre, on obtient de nombreux affichages. En effet, à chaque mouvement de la souris, un évènement `mouseMoved` se produit. Quand le déplacement de la souris est rapide, on obtient des évènements correspondant à des points d'origine espacés :

```
moved : (60,272)
moved : (62,256)
moved : (70,242)
moved : (86,226)
moved : (110,214)
moved : (154,202)
moved : (202,190)
```

Si par contre le mouvement est lent, on obtient un évènement pour chaque déplacement d'un pixel :

```
moved : (160,269)
moved : (161,269)
moved : (162,269)
moved : (162,268)
moved : (162,267)
moved : (163,267)
moved : (163,266)
moved : (164,266)
```

Si on bouge la souris tout en maintenant le bouton enfoncé, on obtient des évènements `mouseDragged`, de façon assez rapide, comme pour les évènements `mouseMoved`. Par exemple :

```
dragged : (116,256) bouton de droite
dragged : (118,255) bouton de droite
dragged : (130,251) bouton de droite
dragged : (148,245) bouton de droite
dragged : (170,239) bouton de droite
dragged : (194,233) bouton de droite
dragged : (214,229) bouton de droite
dragged : (232,229) bouton de droite
```

REMARQUE

Le mécanisme de gestion des évènements basé sur les guetteurs est utilisé de façon assez universelle par `Java`. Nous l'avons mis en oeuvre par exemple pour obtenir la fin d'un programme en réaction à la fermeture de la fenêtre principale, à la section 10.1.3.

10.5 Un exemple complet

10.5.1 Approche modèle/vue/contrôleur

La réalisation d'une application graphique est assez délicate en général. C'est pourquoi il est conseillé d'adopter l'approche modèle/vue/contrôleur. Le principe de cette approche est simplement de séparer en trois parties l'application :

1. **le modèle** : il s'agit de l'objet à représenter. Dans la pratique, on peut utiliser bien sûr plusieurs objets, mais la règle d'or est la suivante : les objets correspondant au modèle ne gèrent ni le graphisme, ni l'interaction avec l'utilisateur.
2. **la vue** : il s'agit de l'objet qui programme la représentation graphique du modèle. En appelant des méthodes du modèle, la vue obtient les informations nécessaires à la représentation. Elle contient donc du code graphique, mais ne gère pas l'interaction avec l'utilisateur.

3. **le contrôleur** : il s'agit de l'objet qui programme l'interaction avec l'utilisateur. Il transforme des évènements en des appels de méthodes modifiant le modèle, et indique à la vue que celui-ci a été modifié.

Dans la pratique, on peut bien entendu avoir plusieurs vues et plusieurs contrôleurs. Le principe fondamental est de bien séparer les trois parties, ce qui permet en général d'obtenir une application relativement simple.

REMARQUE

Dans les exemples précédents faisant intervenir un cadre, la distinction entre le modèle et la vue n'est pas fait, car le cadre n'a pas de sens en dehors du dessin. Ce cas est assez rare dans la pratique.

10.5.2 Le modèle

Dans notre exemple, le modèle est tout simplement un cercle, représenté par un objet de type `Cercle`, dont voici le code :

Cercle

```
1  public class Cercle {
2    public int x;
3    public int y;
4    public int rayon;
5    public Cercle(int u,int v,int r) {
6      x=u;
7      y=v;
8      rayon=r;
9    }
10 }
```

Les variables d'instance sont ici publiques car cela permet une utilisation plus pratique du cercle. La classe `Cercle` est très simple et ne contient pas une ligne de code se rapportant au graphisme.

10.5.3 La vue

La vue est réduite à la plus simple expression : elle se contente de représenter le cercle (sous forme d'un disque), comme l'indique son code :

DessinCercle

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class DessinCercle extends JPanel {
4    private Cercle cercle;
5    public DessinCercle(Cercle c) {
6      cercle=c;
7    }
8    public void paintComponent(Graphics g) {
9      super.paintComponent(g);
10     g.fillOval(cercle.x-cercle.rayon,
11                cercle.y-cercle.rayon,
12                2*cercle.rayon,
13                2*cercle.rayon);
14   }
15 }
```

On voit que la vue ne contient pas une ligne de code correspondant à une gestion de l'interaction avec l'utilisateur.

10.5.4 Les contrôleurs

Dans l'application envisagée, on propose deux contrôleurs : l'un d'entre eux permet d'utiliser le clavier pour modifier le rayon du cercle, alors que l'autre permet d'utiliser la souris pour modifier le centre du cercle. Commençons par le clavier :

ControleCercleRayon

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class ControleCercleRayon extends KeyAdapter {
4    private Cercle cercle;
5    private DessinCercle dessin;
6    public ControleCercleRayon(Cercle c,DessinCercle d) {
7      cercle=c;
8      dessin=d;
9    }
10   public void keyPressed(KeyEvent e) {
11     char c=e.getKeyChar();
12     if(c=='+') {
13       cercle.rayon++;
14       dessin.repaint();
15     } else if(c=='-') {
16       if(cercle.rayon>0) {
17         cercle.rayon--;
18         dessin.repaint();
19       }
20     }
21   }
22 }
```

Il s'agit d'un guetteur de clavier classique. Ses variables d'instance lui permettent de stocker le modèle (l'objet de type `Cercle`) ainsi que la vue (l'objet de type `DessinCercle`). Cela lui permet de modifier le modèle (c'est le code de `keyPressed`) et d'indiquer ensuite à la vue que le modèle a été modifié (en fait, le contrôleur demande simplement à la vue de se réafficher, grâce à la méthode `repaint`).

Le contrôleur de souris fonctionne d'une façon assez similaire, comme l'indique son code :

ControleCerclePosition

```
1  import java.awt.*;
2  import java.awt.event.*;
3  public class ControleCerclePosition extends MouseAdapter {
4    private Cercle cercle;
5    private DessinCercle dessin;
6    public ControleCerclePosition(Cercle c,DessinCercle d) {
7      cercle=c;
8      dessin=d;
9    }
10   public void mouseClicked(MouseEvent e) {
11     cercle.x=e.getX();
```

```
12      cercle.y=e.getY();
13      dessin.repaint();
14    }
15  }
```

Quant un évènement `mouseClicked` se produit, le contrôleur déplace le cercle à l'emplacement du clic de souris, puis réaffiche la vue.

10.5.5 L'application

Il nous faut maintenant assembler les éléments que nous venons de produire, ce qui est relativement simple :

DemoCercle

```
1  import javax.swing.*;
2  import java.awt.*;
3  public class DemoCercle {
4    public static void main(String[] args) {
5      Cercle cercle=new Cercle(20,20,5);
6      JFrame fenêtre=new JFrame("Gestion d'un cercle");
7      fenêtre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8      DessinCercle dessin=new DessinCercle(cercle);
9      dessin.setPreferredSize(new Dimension(200,250));
10     fenêtre.getContentPane().add(dessin,BorderLayout.CENTER);
11     fenêtre.pack();
12     fenêtre.setVisible(true);
13     ControleCerclePosition ccp=new ControleCerclePosition(cercle,dessin);
14     dessin.addMouseListener(ccp);
15     ControleCercleRayon ccr =new ControleCercleRayon(cercle,dessin);
16     dessin.addKeyListener(ccr);
17     dessin.requestFocus();
18   }
19 }
```

La seule "astuce" de ce programme est l'utilisation cruciale des effets de bord. Le programme fonctionne uniquement parce que le cercle créé à la ligne 5 est modifié par effet de bord par les deux contrôleurs, et que l'effet des modifications est observé par la vue. La figure 10.5 représente les liens entre les différents objets qui font fonctionner l'application.

10.6 Représentation d'objets mathématiques

10.6.1 Principe

Le repère utilisé par Java (et en général par tous les langages de programmation) n'est pas un repère usuel, en particulier à cause de son orientation. On souhaite souvent décrire des objets mathématiques (par exemple le graphe d'une fonction) dans un repère usuel, mais donner tout même une représentation graphique de ces objets. Pour ce faire, on doit faire un changement de repère.

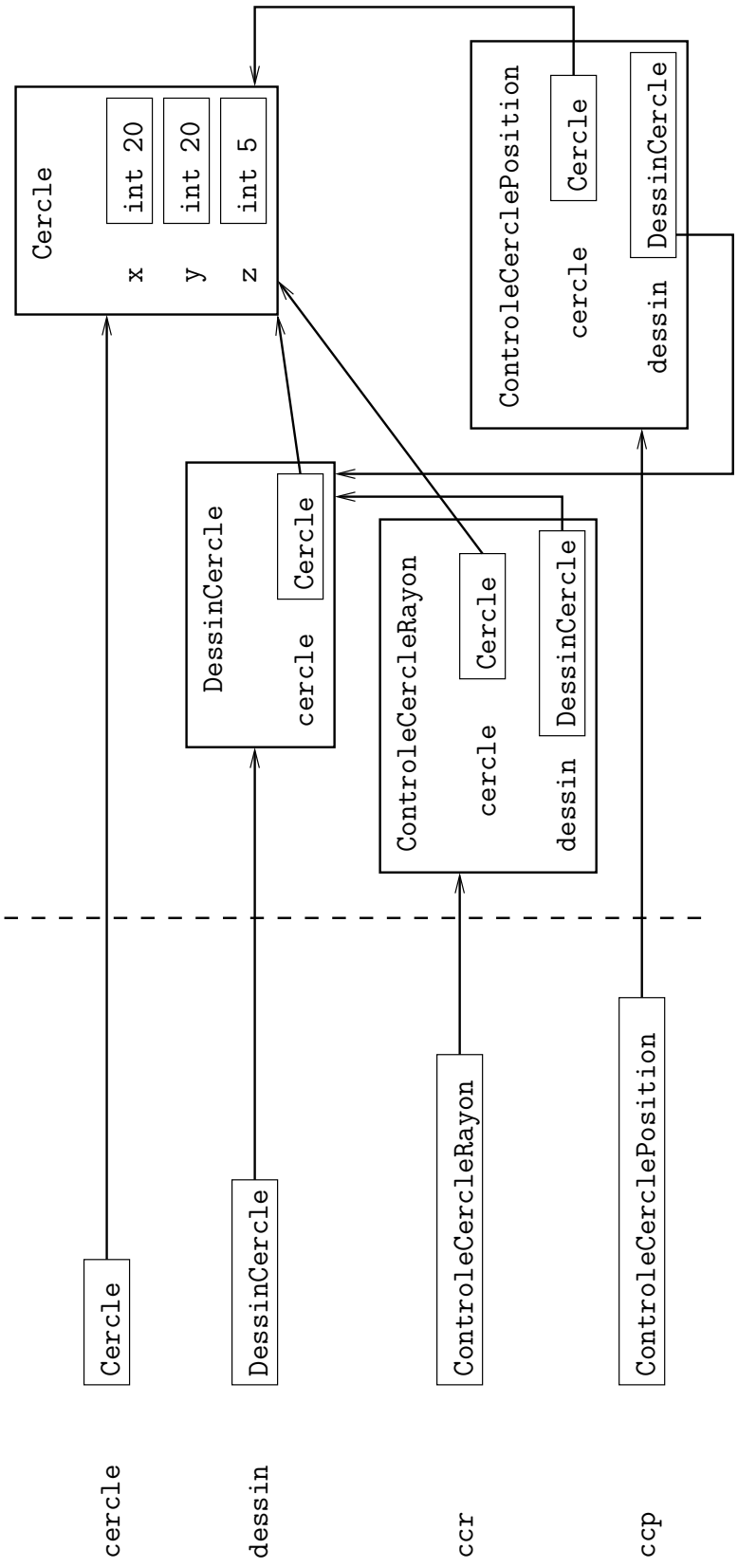

FIG. 10.5 – Les objets en mémoire pour l'application `DemoCercle`

10.6.2 Théorie

On considère un repère orthonormé mathématique usuel (axe des abscisses orienté vers la droite, axe des ordonnées orienté vers le haut). L'écran de l'ordinateur peut représenter une certaine portion du plan. Deux stratégies sont alors possibles :

1. on fixe le rapport de reproduction : on décide par exemple qu'une unité mathématique sera représentée par `x` pixels à l'écran ;
2. on fixe la zone représentée : on définit un rectangle du plan qui occupera toujours la fenêtre entière (et changera donc de rapport de reproduction selon la taille de la fenêtre).

Nous traiterons ici la deuxième solution. Il s'agit donc de représenter une zone mathématique décrite par les points $(x_{\min}, y_{\min})$ et $(x_{\max}, y_{\max})$, respectivement coin inférieur gauche et coin supérieur droit du rectangle, dans une fenêtre. Pour ce faire, on doit convertir les coordonnées depuis le repère mathématique vers le repère de l'écran.

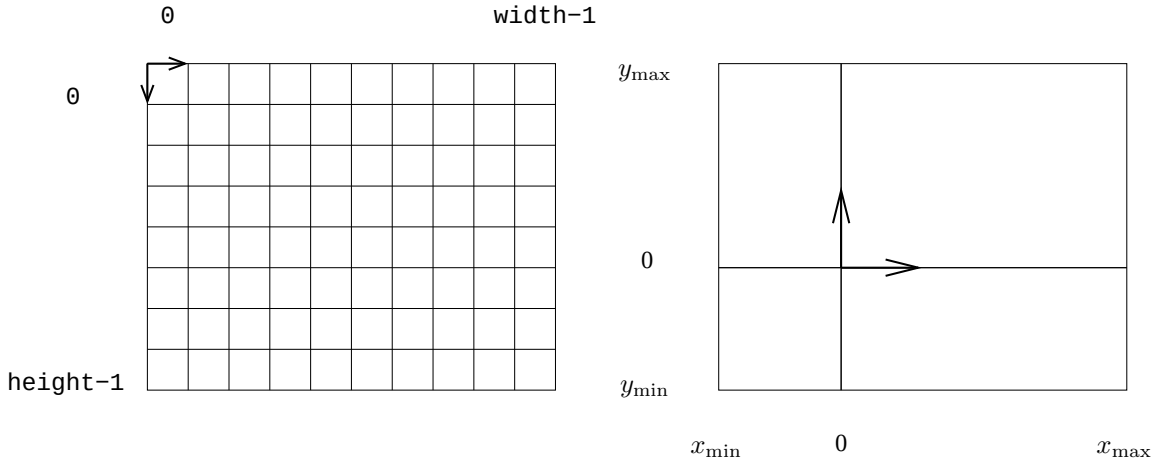

FIG. 10.6 – Repères écran et mathématique

Le changement de repère est une opération affine, qui peut donc se décomposer en une translation suivie d'une opération linéaire (le changement de base qui suit le changement de centre du repère). Cherchons d'abord à identifier la translation.

Le repère écran est centré sur le coin supérieur gauche de la fenêtre. De ce fait, le point de coordonnées écran `(0,0)` est en fait le point de coordonnées mathématiques $(x_{\min}, y_{\max})$ (le coin supérieur gauche de la zone à représenter). La translation de changement de repère doit donc envoyer $(x_{\min}, y_{\max})$ vers `(0,0)`. Il faut donc ajouter $(-x_{\min}, -y_{\max})$ au vecteur coordonnées d'un point exprimé dans le repère de départ.

La transformation linéaire consiste ensuite à passer de la base mathématique vers la base écran. Or, il est clair qu'une abscisse mathématique devient une abscisse écran, sans intervention de son ordonnée mathématique : en d'autres termes, les transformations se font axe par axe.

Considérons donc l'axe des abscisses. Le vecteur qui relie le coin supérieur gauche de la fenêtre à son coin supérieur droit a une coordonnée en x égale à $x_{\max} - x_{\min}$ dans la base mathématique et à `width-1` dans la base écran. La transformation linéaire de l'axe des abscisses doit donc transformer $x_{\max} - x_{\min}$ en `width-1`. Or, une transformation linéaire de $\mathbb{R}$ dans $\mathbb{R}$ se réduit à une simple multiplication. De ce fait, c'est ici la fonction $f(u) = \frac{\texttt{width-1}}{x_{\max} - x_{\min}} u$.

De la même façon, on montre que la transformation sur l'axe des ordonnées est donnée par la fonction $g(v) = \frac{\texttt{height-1}}{y_{\min} - y_{\max}} v$.

En composant les deux transformations, on obtient donc les coordonnées écran en fonction des coordonnées mathématiques par :

$$\begin{aligned} \texttt{X} &= \frac{\texttt{width-1}}{x_{\max} - x_{\min}}(x - x_{\min}) \\ \texttt{Y} &= \frac{\texttt{height-1}}{y_{\min} - y_{\max}}(y - y_{\max}) \end{aligned}$$

10.6.3 Mise en œuvre élémentaire

Voici maintenant un exemple d'objet graphique qui trace le segment rejoignant les points $(-1, 0)$ et $(2, -1)$. On suppose que le `JPanel` doit représenter le pavé définit par $[-3, 3] \times [-2, 1]$. On obtient la classe suivante :

DessinSegment

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class DessinSegment extends JPanel {
4    private double xmin=-3;
5    private double xmax=3;
6    private double ymin=-2;
7    private double ymax=1;
8    public int xMathToGraph(double x) {
9      // attention à la conversion en int qui est obligatoire
10     return (int) ((getWidth()-1)*(x-xmin)/(xmax-xmin));
11   }
12   public int yMathToGraph(double y) {
13     return (int) ((getHeight()-1)*(y-ymax)/(ymin-ymax));
14   }
15   public void paintComponent(Graphics g) {
16     super.paintComponent(g);
17     g.drawLine(xMathToGraph(-1),yMathToGraph(0),
18                xMathToGraph(2),yMathToGraph(-1));
19   }
20 }
```

Les variables `xmin`, `xmax`, `ymin` et `ymax` décrivent la zone mathématique à représenter alors que la taille du `JPanel` est obtenue dynamiquement grâce à un appel classique aux méthodes `getHeight` et `getWidth` dans les méthodes de conversion. Si on place cet objet graphique dans une fenêtre, on obtient comme affichage un segment. Si on change la taille de la fenêtre, le dessin est réaffiché avec la nouvelle échelle : le segment conserve la même position relative par rapport au cadre, ainsi que les mêmes proportions, comme l'illustre la figure 10.7.

10.6.4 Représentation du graphe d'une fonction

Le problème

Le graphe d'une fonction f de $[a, b] \subset \mathbb{R}$ dans $\mathbb{R}$ est l'ensemble des points $(x, f(x))$ pour $x \in [a, b]$. Il est bien entendu impossible de représenter tel que cet ensemble qui est infini. La représentation informatique passe donc par une approximation. Pour ce faire, on divise l'intervalle $[a, b]$ en choisissant n points régulièrement espacés (par exemple), avec $x_0 = a$ et $x_n = b$. Ensuite, on trace les segments reliant $(x_i, f(x_i))$ à $(x_{i+1}, f(x_{i+1}))$ pour tout $i < n$. La ligne brisée obtenue est une approximation du graphe de f.

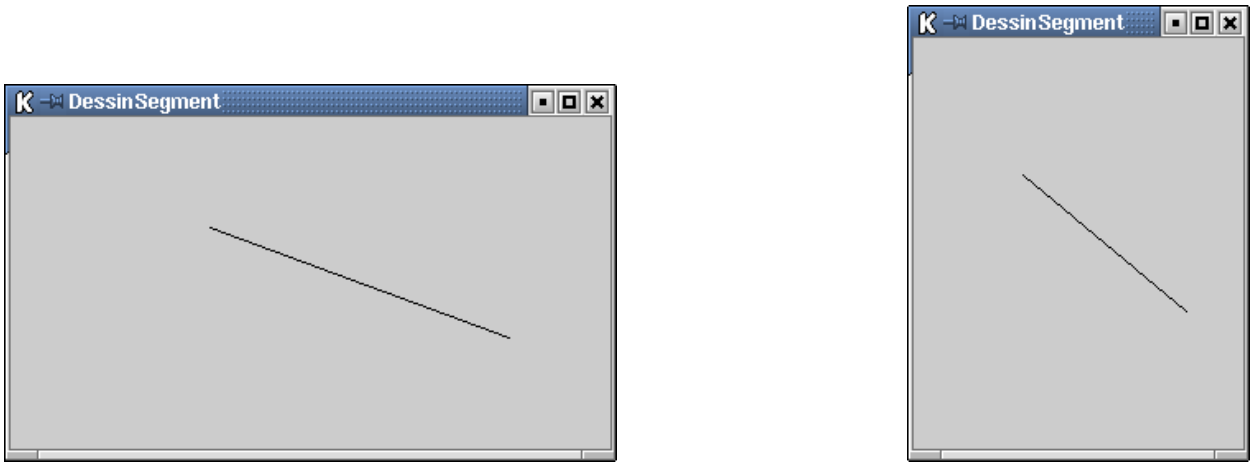

FIG. 10.7 – Affichage produit par `ToileSegment`

Solution simple

Pour obtenir une solution simple, on fixe la valeur de n, sous forme d'une variable d'instance de la toile. Considérons par exemple le tracé de la fonction $\cos(x^2)$ sur l'intervalle $[0, \pi]$. Voici une classe proposant le tracé de cette fonction :

DessinCos

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class DessinCos extends JPanel {
4    // la fonction à tracer (ici, cos(x^2))
5    public static double f(double x) {
6      return Math.cos(x*x);
7    }
8    private double xmin=0;
9    private double xmax=Math.PI;
10   private double ymin=-1;
11   private double ymax=1;
12   private int n=10;
13   public int xMathToGraph(double x) {
14     return (int) ((getWidth()-1)*(x-xmin)/(xmax-xmin));
15   }
16   public int yMathToGraph(double y) {
17     return (int) ((getHeight()-1)*(y-ymax)/(ymin-ymax));
18   }
19   public void paintComponent(Graphics g) {
20     super.paintComponent(g);
21     double x=xmin;
22     double f=f(x);
23     // n points, soit n-1 intervalles
24     double h=(xmax-xmin)/(n-1);
25     for(int i=1;i<n;i++) {
26       double xi=xmin+i*h;
27       double fi=f(xi);
```

```
28        g.drawLine(xMathToGraph(x),yMathToGraph(f),
29                   xMathToGraph(xi),yMathToGraph(fi));
30        x=xi;
31        f=fi;
32      }
33    }
34  }
```

Cette classe est relativement simple, la boucle de la méthode `paintComponent` se chargeant du tracé. Le résultat graphique obtenu est assez mauvais, comme le montre la figure 10.8. Le problème

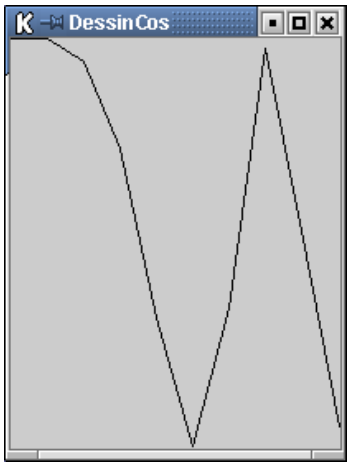

FIG. 10.8 – Affichage produit par `DessinCos(n=10)`

est simplement que le nombre de points utilisés est trop faible pour donner une représentation correcte. Si on passe à `n=30`, on obtient une représentation plus conforme à la réalité (voir la figure 10.9), mais le tracé devient un peu plus lent.

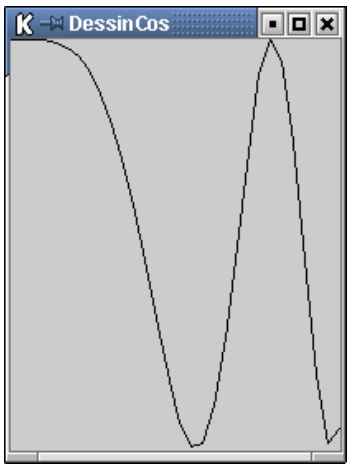

FIG. 10.9 – Affichage produit par `DessinCos(n=30)`

Solution optimale

La solution proposée par `DessinCos` est intéressante, mais pas optimale. En effet, on doit faire plusieurs essais avant de déterminer la bonne valeur pour `n`, c'est-à-dire le nombre de segments à utiliser. Or, dans la pratique, il existe un nombre de segments optimal qui dépend des dimensions de la toile. En effet, comme nous l'avons déjà indiqué à la section 10.2.1, l'ordinateur ne travaille pas sur des points sans dimension, mais sur des pixels. De ce fait, quand la toile dans laquelle on dessine comporte par exemple `w` colonnes de pixels, il est inutile de découper l'intervalle $[a, b]$ en plus de `w` segments. Par contre, si on utilise moins de `w` segments, on ne profite pas de toute la précision offerte par l'écran. De ce fait, le nombre de segments optimal est `w`.

La méthode la plus simple permettant d'utiliser `w` segments consiste à remplacer les utilisations de `n` par `getWidth()` dans le code de `DessinCos`. On peut aussi remarquer que le passage par les coordonnées mathématiques ne se fait pas de façon très logique dans le code actuel. En effet, il serait plus naturel de travailler directement en coordonnées écran pour les abscisses. Il faut alors transformer une abscisse écran en abscisse mathématique, ce qui ne pose pas de problème : il suffit d'inverser la formule de la section 10.6.2. On obtient le code suivant :

DessinCosOptimal

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class DessinCosOptimal extends JPanel {
4    // la fonction à tracer (ici, cos(x^2))
5    public static double f(double x) {
6      return Math.cos(x*x);
7    }
8    private double xmin=0;
9    private double xmax=Math.PI;
10   private double ymin=-1;
11   private double ymax=1;
12   public double xGraphToMath(int x) {
13     return xmin+((xmax-xmin)*x)/(getWidth()-1);
14   }
15   public int yMathToGraph(double y) {
16     return (int) ((getHeight()-1)*(y-ymax)/(ymin-ymax));
17   }
18   public void paintComponent(Graphics g) {
19     super.paintComponent(g);
20     double f=f(xmin);
21     for(int i=0;i<getWidth();i++) {
22       double fi=f(xGraphToMath(i+1));
23       g.drawLine(i,yMathToGraph(f),i+1,yMathToGraph(fi));
24       f=fi;
25     }
26   }
27 }
```

Comme le montre la figure 10.10, le tracé obtenu est beaucoup plus satisfaisant (plus "lisse"). De plus, le tracé reste "parfait" quelles que soient les dimensions effective de la fenêtre.

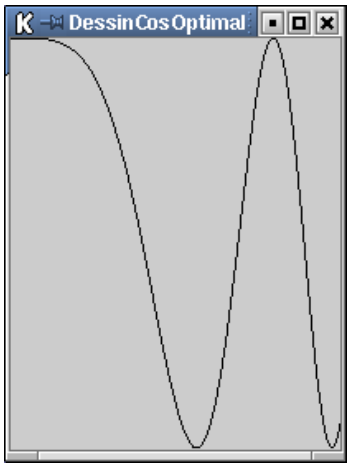

FIG. 10.10 – Affichage produit par `DessinCosOptimal`

Bibliographie

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilateurs Principes, techniques et outils*. InterEditions, Paris, 1989.

[2] Alfred Aho and Jeffrey Ullman. *Concepts fondamentaux de l'informatique*. Dunod, 1993.

[3] Mary Campione and Kathy Walrath. *The Java Tutorial Second Edition*. Addison-Wesley, 2000. Disponible à l'URL http://java.sun.com/docs/books/tutorial/index.html.

[4] Bruce Eckel. *Thinking in JAVA*. Prentice-Hall, 1998.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Java Series. Addison-Wesley, August 1996. Version 1.0, disponible à l'URL http://java.sun.com/docs/books/jls/index.html.

[7] Mark Grand. *JAVA Language Reference*. O'REILLY, second edition, July 1997.

[8] Tim Lindholm and Frank Yelling. *The Java Virtual Machine Specification*. Java Series. Addison-Wesley, September 1996. Disponible à l'URL http://java.sun.com/docs/books/vmspec/index.html.

[9] Fabrice Rossi. *Initiation à la programmation Java*. Université Paris-IX Dauphine, 1997.

[10] Fabrice Rossi. *Programmation Objet*. Université Paris-IX Dauphine, 1999.

[11] Sun Microsystems. *Code Conventions for the Java Programming Language*, April 1999. Disponible à l'URL http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html.

[12] Sun Microsystems. *Java 2 SDK, Standard Edition Documentation (version 1.3)*, 2000. Disponible à l'URL http://java.sun.com/j2se/1.3/docs/index.html.

[13] Andrew Tanenbaum. *Systèmes d'exploitation*. Prentice Hall & InterEditions, 1994.

Index