2001-2002

UFR Mathématiques de la Décision

Première Année de DEUG Sciences et Technologie mention MASS

Informatique

Volume I

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.

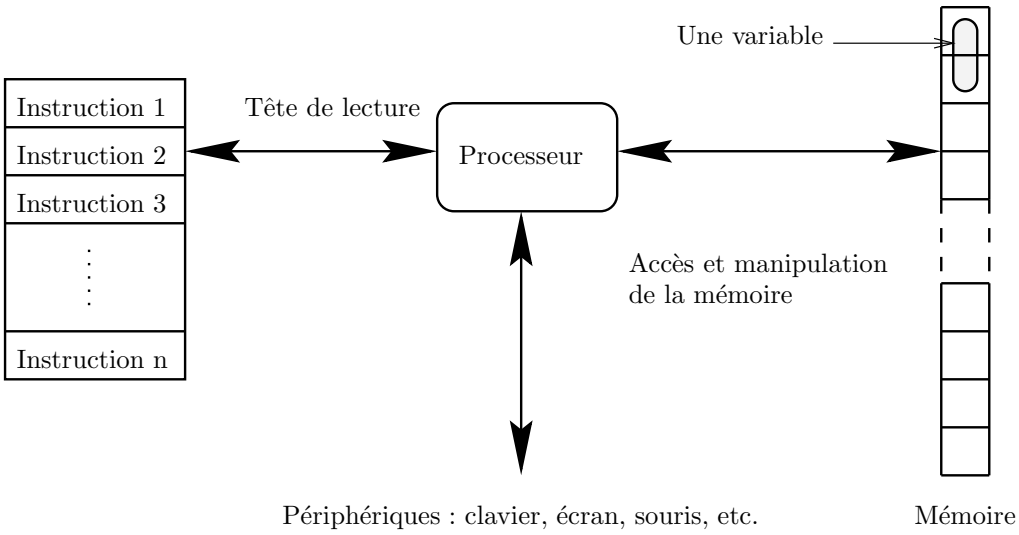

FIG. 1.1 – L'ordinateur abstrait

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**.

[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.

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 (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

[5] Le lecteur intéressé pourra se reporter à [1], ouvrage de référence sur la théorie des langages informatiques.

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 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.

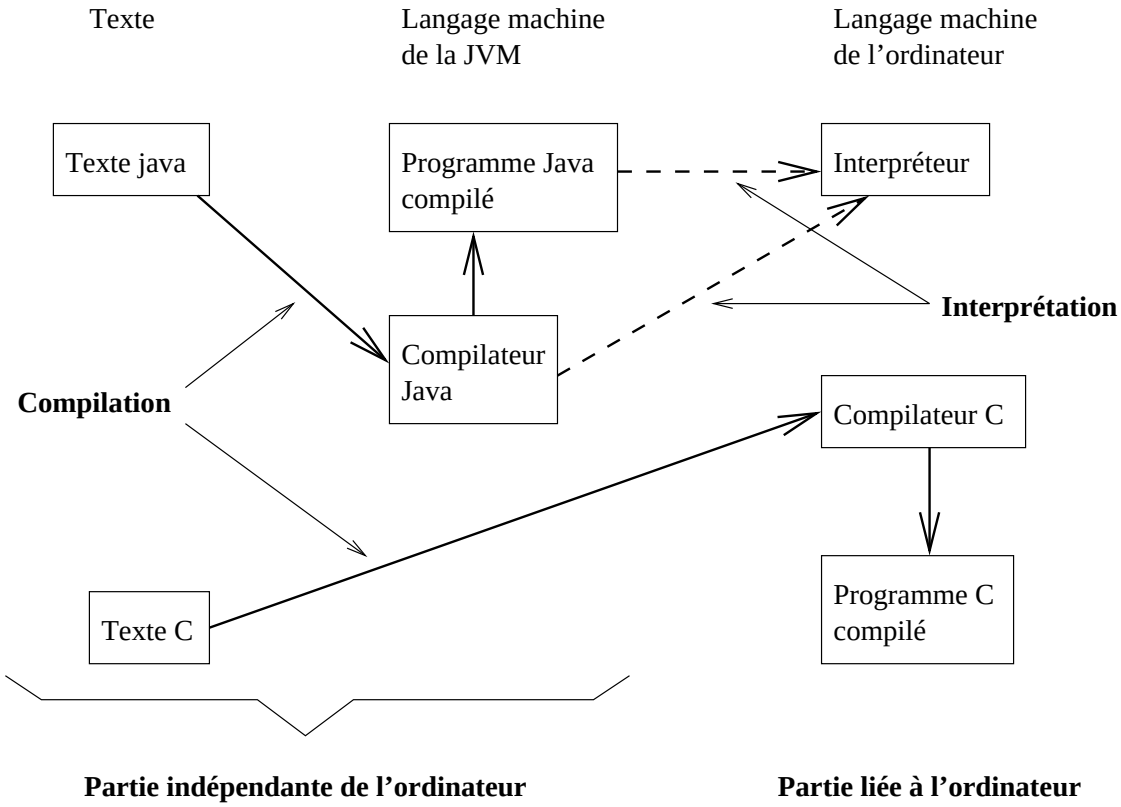

FIG. 1.2 – Compilation et interprétation

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 :

[6] Les Macintosh récents utilisent des processeurs PowerPC.
[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.

- 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 ;
- 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* `_`.

[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.

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.

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 7). 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.

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

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.

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 en `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] :

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

```
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.

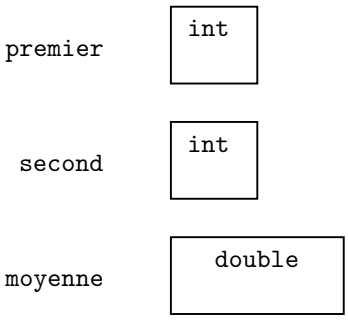

FIG. 2.1 – Déclaration de variables

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 créée pour chaque identificateur. La forme générale est ainsi :

type identificateur_1, identificateur_2,..., identificateur_n ;

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.

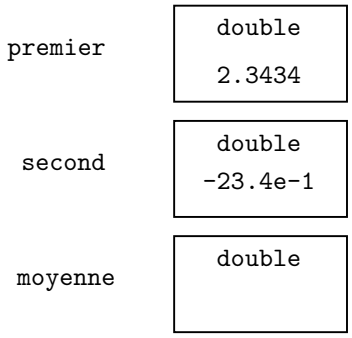

FIG. 2.2 – Affectation

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 seule 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

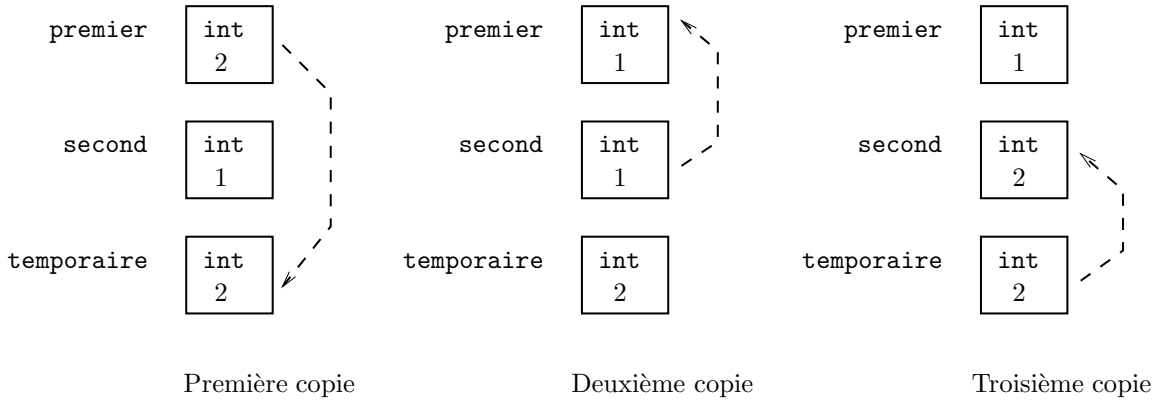

FIG. 2.3 – Illustration de l'échange de deux variables

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 `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 déclaré 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;
```

[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.

```
      ^
1 error
```

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 intéressés 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 compatibles 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 `float`s, 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 `double`s, 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 dû 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 priorités). 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 :

```
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 18), 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` :

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

[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 $[-3.5] = -4$.

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 ;`

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

[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`.

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);
```

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 9 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 7 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 leurs 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 :

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

`double sin(double)`
: Calcule et renvoie le sinus de son paramètre.

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.

[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.

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

`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` choisi 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 devine 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 9. 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 suivants :

```
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.

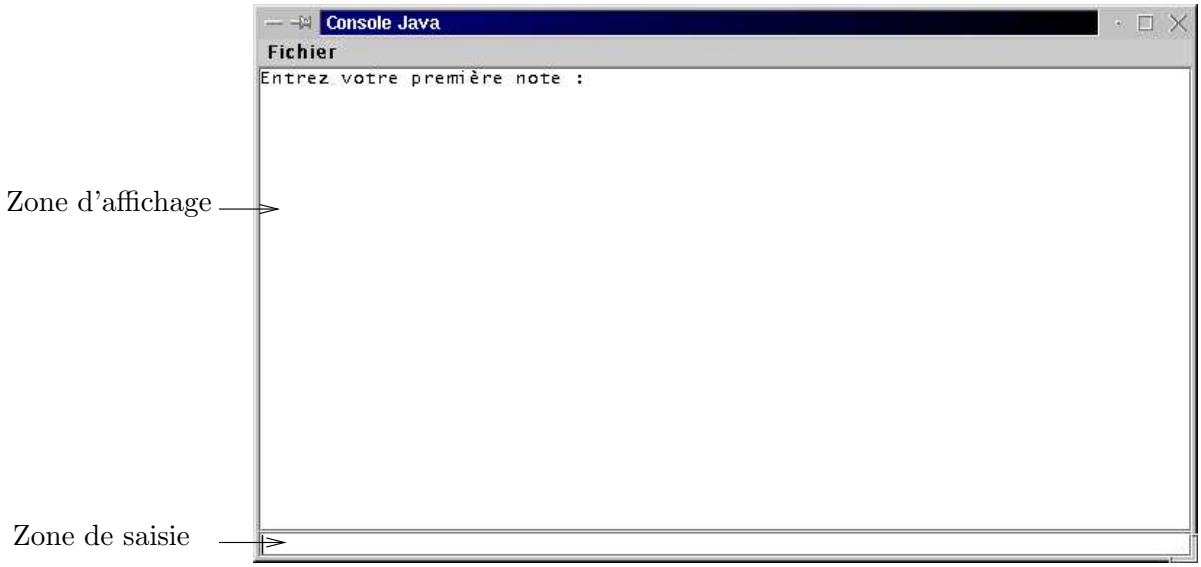

FIG. 3.1 – La console après ouverture et affichage

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 :

[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.

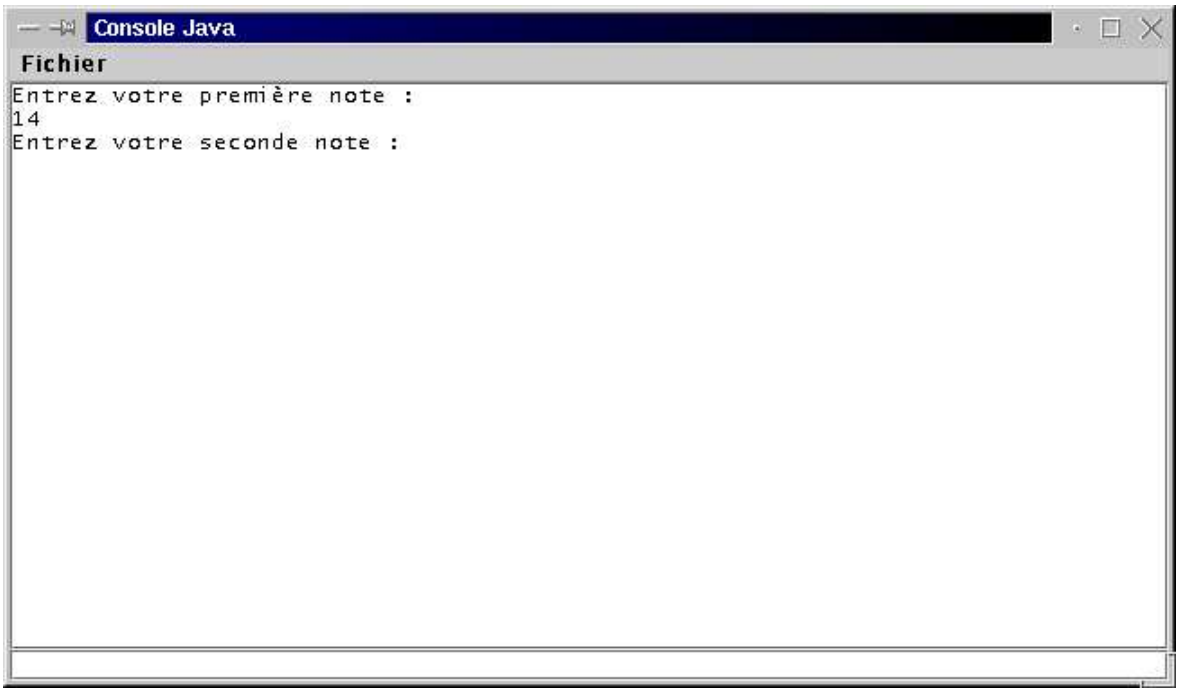

FIG. 3.2 – La console après une saisie

CalculMoyenneSaisie

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

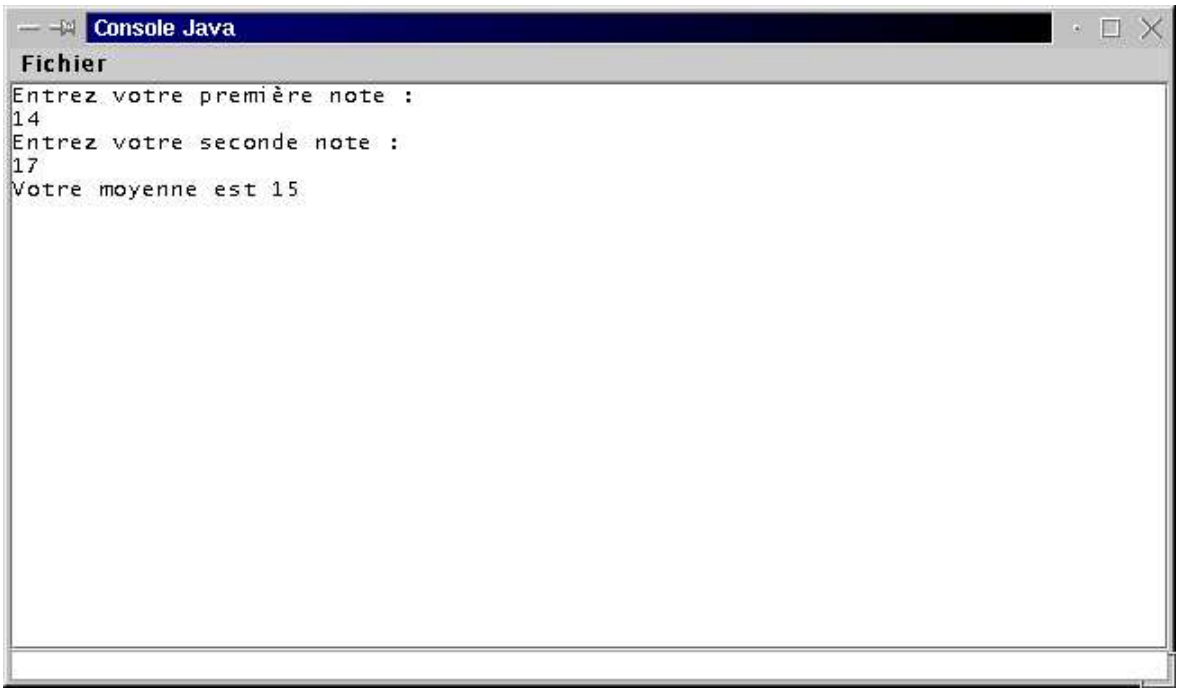

FIG. 3.3 – La console après les deux saisies

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` :

`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 autres) 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 méthode.
- 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 :

[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.

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;
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:1: package dauphine.util does not exist
import dauphine.util.*;
^
PasDeVariable.java:4: cannot resolve symbol
symbol  : variable Console
location: class PasDeVariable
    Console.start();
    ^
PasDeVariable.java:5: cannot resolve symbol
symbol  : variable Console
location: class PasDeVariable
    double x=Console.readDouble();
             ^
PasDeVariable.java:11: cannot resolve symbol
symbol  : variable signe
location: class PasDeVariable
    System.out.println(signe);
                       ^
4 errors
```

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:1: package dauphine.util does not exist
import dauphine.util.*;
^
PasDeValeur.java:4: cannot resolve symbol
symbol  : variable Console
location: class PasDeValeur
    Console.start();
    ^
PasDeValeur.java:5: cannot resolve symbol
symbol  : variable Console
location: class PasDeValeur
    double x=Console.readDouble();
             ^
3 errors
```

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 prises 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

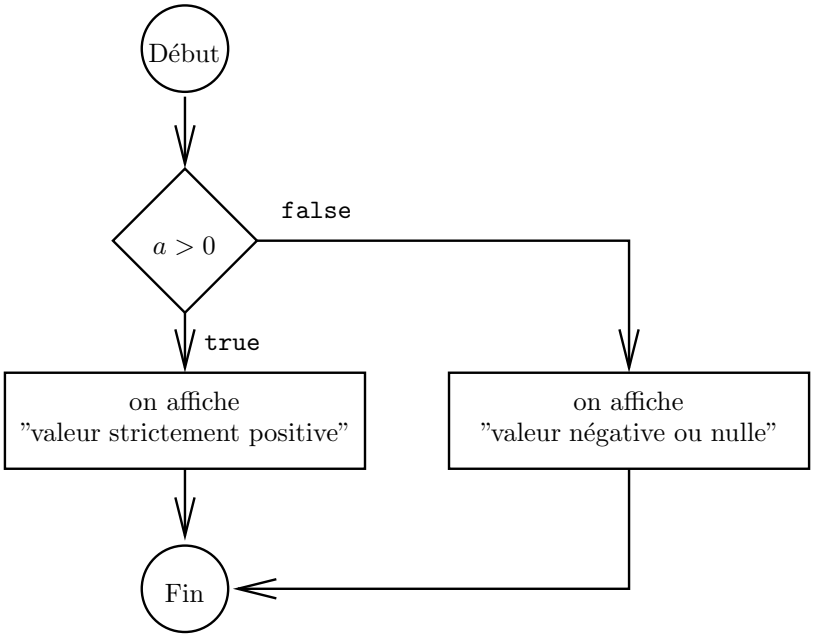

FIG. 4.1 – Organigramme simple

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.*

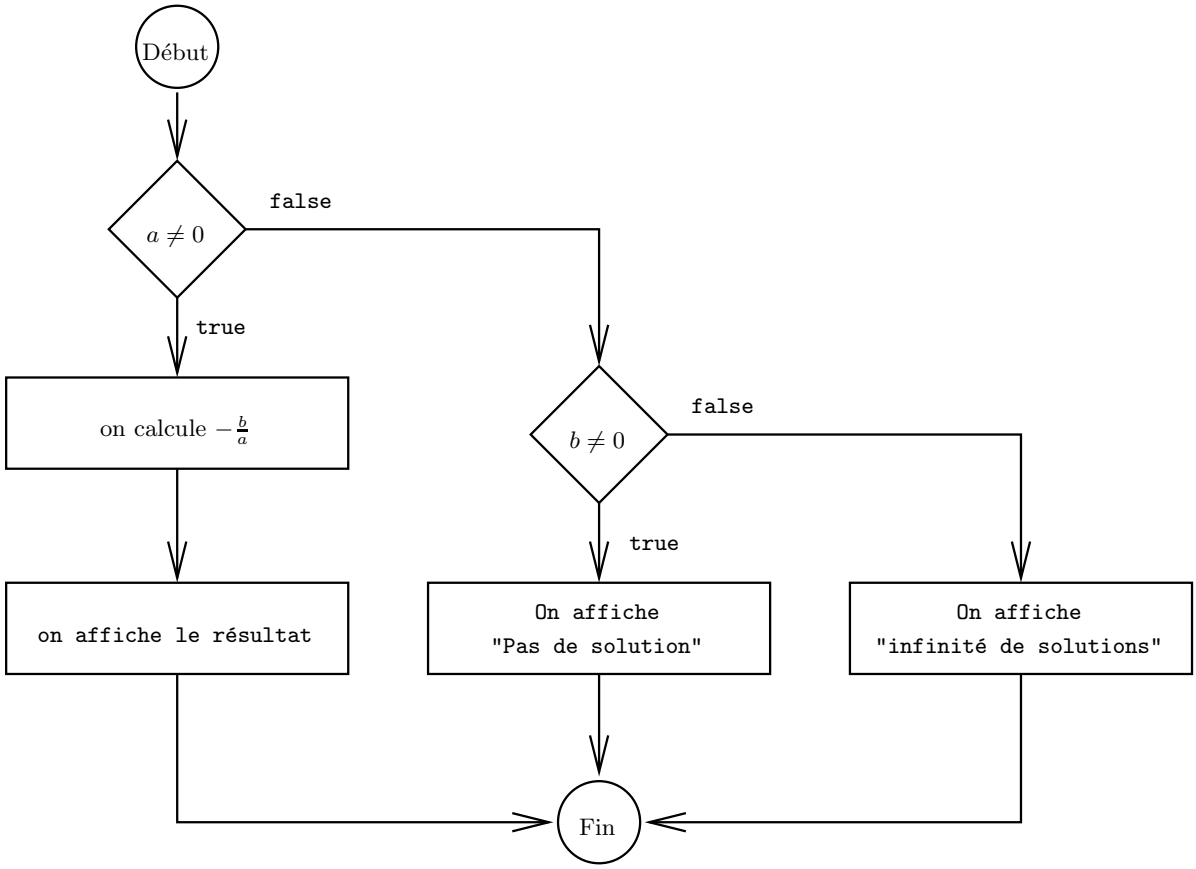

FIG. 4.2 – Organigramme de résolution de $ax + b = 0$

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;
```

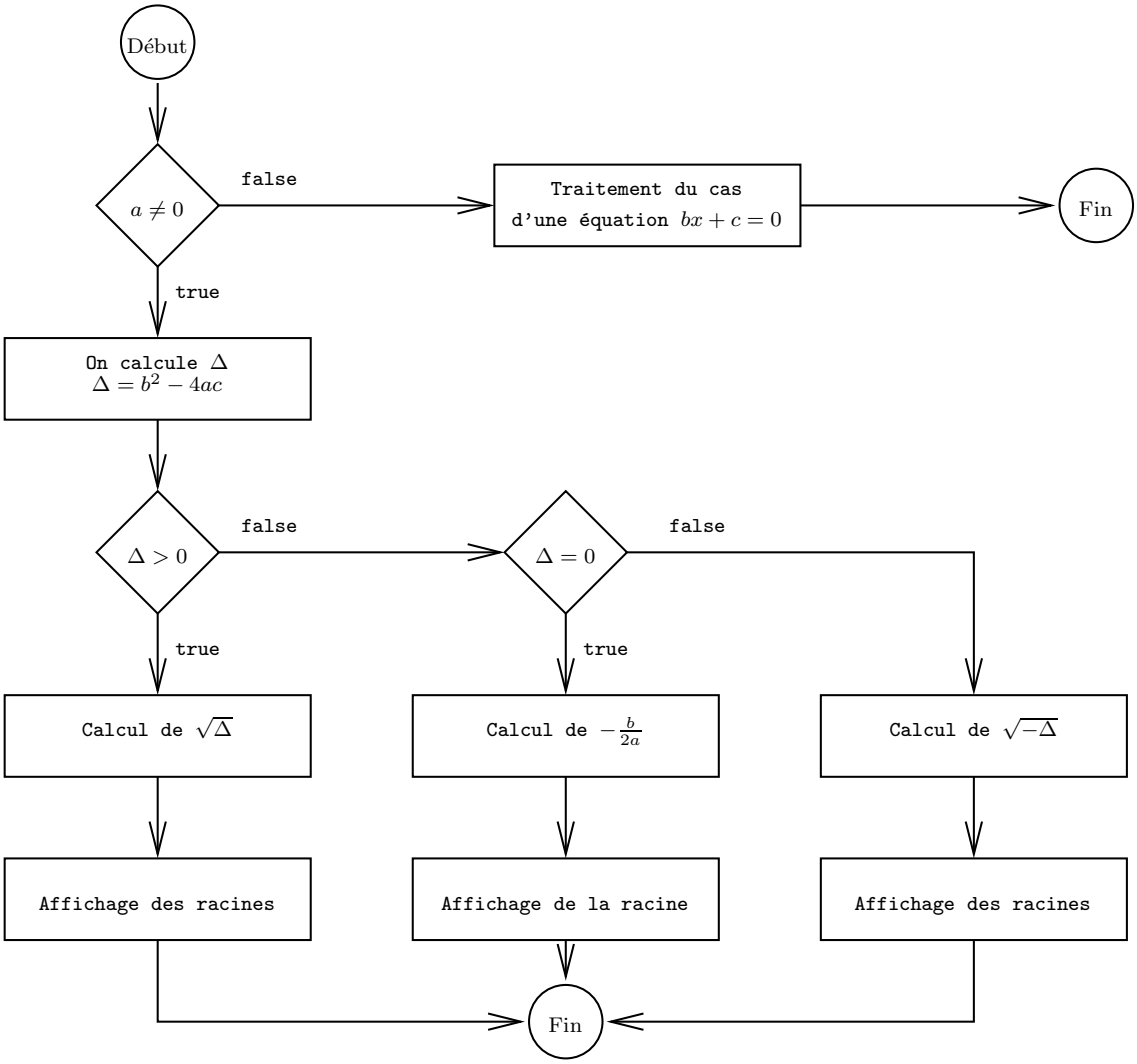

FIG. 4.3 – Organigramme de résolution de $ax^2 + bx + c = 0$

```
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 possibilités 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 `switch` : 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** 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.