MCours.com
Le N°1 du cours et exercices sur Internet

NFA001 – Algoritmique et programmation avec Java : notions de base

Notes de cours

Chap. 1 – Introduction
Chap. 2 – L'essentiel pour commencer
Chap. 3 – Exécution des programmes
Chap. 4 – Types, expressions et typage
Chap. 5 – Compléments sur l'itération
Chap. 6 – Tableaux
Chap. 7 – Sous-programmes
Chap. 8 – Exceptions
Chap. 9 – Courte introduction au type String

Chapitre 1

Introduction

Nos objectifs pédagogiques sont :

1. Étudier les concepts de base de la programmation dans les langages de haut-niveau, de manière à :
   - les appliquer en Java,
   - comprendre des concepts présents dans beaucoup d'autre langages de programmation.
2. S'initier à l'analyse et resolution de problèmes, via la programmation,
3. Acquerir certaines méthodes de resolution des problèmes classiques en informatique.

1.1 Pourquoi Java ?

Tout langage de programmation de haut niveau renferme *grosso modo* les mêmes concepts. Chacun devient en quelque sorte, un *dialecte* différent pour un *pouvoir d'expression* (ce que l'on peut faire avec) très proche. Notre but est, qu'une fois les concepts de base de la programmation acquis via ce cours, et via la la pratique de Java, vous soyez capables de :

- vous former par vous même dans l'apprentissage d'un autre langage de programmation (dialecte)
- de programmer rapidement (parler ce dialecte) en gardant les mêmes techniques et reflexes déjà appris avec l'étude de Java.

Les avantages de Java :

- Il existe des environnements de développement gratuits que vous pouvez installer par vous mêmes,
- Java est fortement typé, ce qui signifie, que beaucoup d'erreurs sont détectée s automatiquement (et c'est essentiel pour un débutant)
- Java incorpore des nombreux traits de programmation de haut niveau : orienté objet, exceptions, polymorphisme, gestion de la mémoire, transparence des pointeurs.
- Java possède une sémantique précise.
- Les programmes Java sont portables. Leur exécution est indépendante de la plateforme d'installation (type de machine). Il suffit de disposer d'un environnement d'exécution Java pour l'exécuter.
- Java est robuste et sécurisé.

Principaux domaines d'application :

– Internet et le Web,
– Programmation distribuée,
– Programmation embarquée.

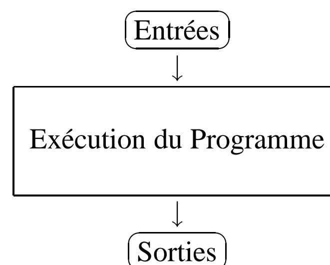
1.2 Les programmes

Les programmes servent à décrire les solutions d'un problème.

Qu'est-ce qu'un programme ?
– C'est la **description d'une méthode** mécanique (i.e, appliquable par une machine) de **resolution** d'un problème donné,
– Cette description est donnée par une **suite d'instructions** d'un langage de programmation.
– Ces instructions ont pour but de **traiter et transformer les données** du problème à resoudre, jusqu'à aboutir à une solution.

Un programme est en quelque sorte une méthode à suivre pour trouver les solutions, mais n'est pas une solution en soi.

Trouver les solutions : **exécuter** le programme sur les données du problème.
– Les traitements décrits par les instructions sont appliqués aux données (*entrées*),
– Les données ainsi transformées permettent d'aboutir aux solutions recherchées (*sorties*).



1.3 Les ordinateurs

Composants principaux :
– **Unité centrale (CPU)** ou processeur, pour le traitement des données et instructions logées en mémoire centrale. Capable d'exécuter un ensemble d'instructions dites *instructions machine*. Il s'agit d'opérations simples telles que la lecture/écriture en mémoire centrale, opérations arithmétiques et logiques, comparaison des valeurs, branchement (saut) vers une adresse afin de poursuivre l'exécution, etc. Ces instructions sont codées en binaire (s'équences de 0 et de 1) et diffèrent dans chaque plateforme matérielle. Ainsi, une instruction qui s'exécute sur un processeur Intel/PC ne peut pas s'exécuter sur un processeur SPARC/Sun.
– **Mémoire Centrale (RAM)**, très rapide (accès directe), mais volatile. Cette mémoire est organisé comment une suite d'emplacements appelés *mots* et munis chacun d'une adresse. On doit y stocker les données à traiter, ainsi que les instructions machine à exécuter. Tout programme à exécuter et toute donnée à traiter doit être chargé en mémoire centrale au préalable.

- **Périphériques**. Dispositifs de communication de l'ordinateur pour l'aquisition, production, communication des données : clavier, écran, enceintes, ports de communication ; et pour le stockage persistent tels que disques durs, disquettes, etc.

Fonctionnalités principales :
- Exécution des instructions des programmes chargés en mémoire centrale.
- Commande des périphériques.
- Acquisition, stockage, communication et production des données : saisie au clavier, affichage à l'écran, etc.

1.4 Les langages de programmation

Il en existe deux groupes principaux :
- **Langages de haut niveau**. (Ex : C, Ada, Pascal, Cobol, Java, OCaml, Python). Ils fournissent des nombreuses constructions sophistiquées qui facilitent l'écriture des programmes. Ils sont comprehensibles par les humains, mais pas directement exécutables par les machines. Un programme écrit en langage de haut niveau devra être traduit en langage machine avant son exécution.
- **Langages de bas niveau**. Ce sont les différents ensembles d'instructions propres à chaque machine (SPARC/Sun, Intel/PC, etc). Appelés également *langages cibles* ou *natifs*. Ils sont codés en binaire et directement exécutables par chaque machine.

Langages de haut niveau

Un langage de programmation est composé de trois ensembles : un ensemble de *types de données*, un ensemble de règles de *construction syntaxiques des instructions* et un ensemble de *règles sémantiques*. Ces trois composants décrivent toutes les possibilités des données et instructions dans un programme, ainsi que leur comportément à l'exécution.

- **Les types des données** : utilisées pour décrire et modéliser les données. *Exemple :* Le type `int` en Java sert à modéliser les nombres entiers.

- **La syntaxe** : règles de *formation textuelle* des instructions et des programmes. *Exemple :* pour écrire l'expression mathématique $1 \leq x \leq 7$, une syntaxe possible en Java est : `1 <= x && x <= 7`.

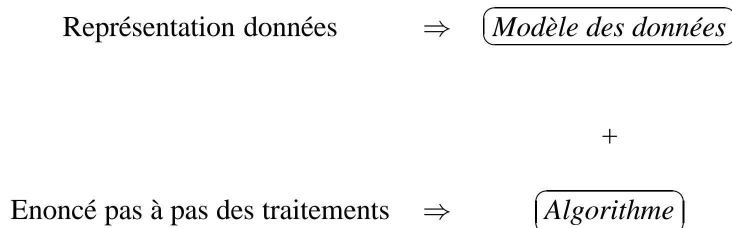
- **La sémantique** : règles qui précisent, d'une part le *comportement* ou le *sens* des contructions syntaxiques lorsqu'elles sont executées par une machine et d'autre part, les *contraintes de cohérence entre types*. *Exemples* : `4+3*2` équivaut en Java à la valeur entière 10. L'expression `"bonjour" * 2` est bien formée du point de vue de la syntaxe, mais pas du point de vue sémantique : `*` est un opérateur numérique, et ici il a pour opérande la chaîne de caractères `"bonjour"`. Cette expression est incohérente vis-à-vis des types.

1.5 La production des programmes

La fabrication d'un programme suit les phases suivantes :

1. **Analyse et Conception** : On établit précisement le problème à resoudre, les données de départ et ce que l'on souhaite obtenir en résultat, puis, on raisonne sur la manière de représenter les

données du problème, et sur une méthode de resolution. Cette méthode est ensuite énoncée sous forme de suite de pas à accomplir pour aboutir aux solutions : c'est *l'algorithme de resolution du problème*.

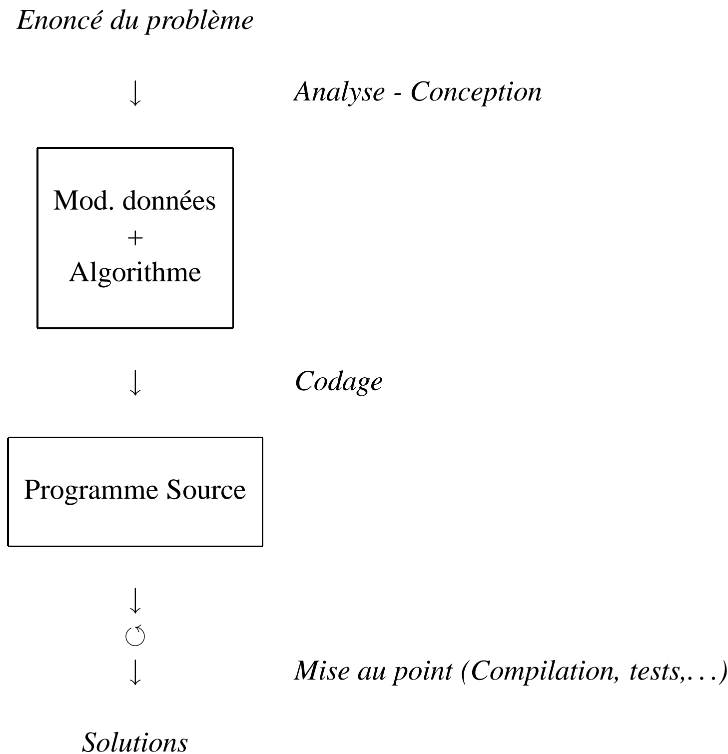


2. **Codage** : il s'agit de traduire l'algorithme en langage de programmation, et sous forme de fichier texte (c.a.d., suite de caractères sans formattage ni mise en page). Le résultat du codage est un fichier appelé *code source* du programme.

3. **Mise au point** : comprend le plus souvent plusieurs étapes repétées jusqu'à ce que le programme semble satisfaisant :

   – Compilation : un langage de programmation de haut niveau n'est pas directement comprehensible par la machine. Il faut donc traduire le *code source* du programme vers le *langage natif* de la machine (ou parfois vers du code intermédiaire). Le résultat est un nouveau fichier écrit en langage machine, et appelé *code objet*. Cette étape comprend souvent également *l'édition de liens*, qui est la préparation du *code objet* pour l'exécution.
   – Tests : exécution du code objet avec divers cas typiques des entrées, ou *jeu de tests*. C'est le moment où la plupart des erreurs apparaisent.
   – Correction d'erreurs : on modifie le code de manière à corriger les erreurs au fil des tests, et l'on recommence la compilation, exécution et tests, etc.

4. **Maintenance** : il s'agit le plus souvent de la correction des erreurs apparues après la mise en service, mais aussi de la modification du programme pour l'adapter à de nouvelles spécifications du problème.



1.6 Traducteurs de programmes

Chaque langage de programmation est équipé d'un *logiciel de traduction* des programmes écrits dans ce langage. Entre autres traitements, il traduit chaque instruction de haut niveau en plusieurs instructions machine équivalentes.

– code source : fichier texte avec les instructions à traduire.
– code cible : fichier (binaire ou texte, selon le type de traduction) résultat de la traduction.
– code objet : fichier binaire avec des instructions machine.
– pseudo-code ou byte-code : fichier texte contenant du code intermédiare d'assez bas niveau, mais non exécutable directement par la machine.

Il existe deux sortes de traducteurs : les **interprètes** et les **compilateurs**.

Compilateurs

Réalisent les traitements suivants sur le code source :

1. *Analyse syntaxique* : vérifie que le programme est correct d'après les règles de syntaxe.
2. *Typage* : vérifie la correction d'après les règles (sémantiques) de cohérence entre types.
3. *Traduction* : S'il n'y pas d'erreurs de syntaxe ni de typage, chaque instruction du programme source est traduite vers plusieurs instructions du langage natif ou de *pseudo-code* (code intermédiare). Le résultat de la compilation est un nouveau fichier dit de *code cible* qui est pour la plupart des compilateurs (Ada, Pascal, etc.) du code machine pour la plupart des compilateurs mais ou du pseudo-code. Dans ce dernier cas, il sera à traduire avant exécution. S'il y a des

erreurs de syntaxe ou de typage, il faut les corriger et relancer la compilation jusqu'à ce qu'il n'y ait plus d'erreurs. Seulement alors, le code cible est généré.

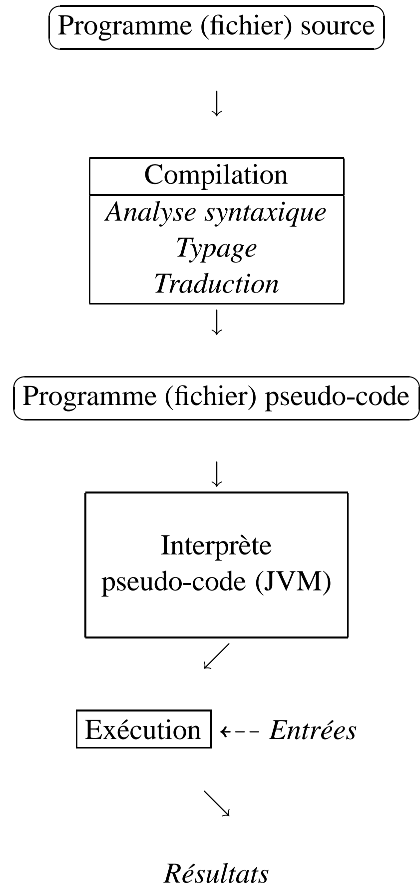
C, Pascal, Ada, Ocaml, Java sont des langages compilés. Mais Ocaml et Java sont compilés vers du pseudo-code, plutôt que vers du code natif.

Interpréteurs

Pour chaque instruction du programme, ils réalisent, l'une après l'autre, traduction en code puis exécution. Ce mécanisme a pour inconvénient de limiter le type d'analyses réalisées sur le programme source, mais surtout, il autorise l'exécution d'un programme même s'il présente des erreurs de syntaxe ou de sémantique. Cependant, ce mécanisme est bien adapté aux programmes pre-compilés vers du pseudo-code. C'est le cas de Java.

Compilation en Java

Un des objectifs de Java, est de permettre l'écriture des programmes independants des plateformes matérielles d'exécution. C'est le defi de la programmation distribuée que d'envoyer des programmes à travers le réseau pour leur exécution à distance. Dans ce contexte, la compilation vers du code natif n'a pas grand sens. C'est pourquoi en Java, on compile vers un code intermédiaire ou *byte-code* identique pour tous les ordinateurs. L'environnement d'exécution Java est équipé, dans chaque machine, d'un interprète qui lui est propre. Il traduit le byte-code vers du code natif et l'exécute, avec un minimum d'analyses. L'ensemble des interprètes est c'est que l'on appelle la Machine Virtuelle Java (JVM). Ainsi, un programme traduit en byte-code, pourra être exécutée par n'importe quelle machine munie d'une JVM. Les programmes Java développés sur une machine particulière sont *portables* sur n'importe quelle installation sans avoir à réecrire du code.



Le programme `Bonjour`

Ce texte correspond à un programme Java syntaxiquement et sémantiquement correct.

Listing 1.1 – (lien vers le code brut)

```
1
2  /* Premier programme
3    - a enregistrer dans le fichier Bonjour.java
4    - a compilar par javac  Bonjour.java
5    - a executer par java  Bonjour
6  */
7
8  public class Bonjour {
9    public static void main (String[] args) {
10     Terminal.ecrireStringln("Bonjour tout le monde !");
11   }
12 }
```

- Le texte entre `/*` et `*/` est un commentaire.
- Le nom après `public class` est le nom du programme. Dans cet exemple, c'est `Bonjour`.
- Ce programme ne manipule qu'une donnée : le message

```
Bonjour tout le monde !
```

qui est affichée à l'écran lors de l'exécution.
- Nous le composons dans un fichier du même nom que le programme, et complété de l'extension `java`, à savoir, dans le fichier `Bonjour.java`

Ce programme n'est pas directement exécutable par l'ordinateur. Nous devons donc le traduire en langage machine.

Compilation et exécution de Bonjour

- Création du fichier source : Le code Java est mis dans un fichier nommé `Bonjour.java`. C'est le programme source.
- Compilation : Nous lançons la commande de compilation sur le fichier `Bonjour.java`,
  ```
  Java/Essais> javac Bonjour.java
  Java/Essais>
  ```
  La compilation reussit (pas d'erreurs). Le pseudo-code obtenu se trouve dans le fichier `Bonjour.class` du repertoire courant, mais il n'est pas directement exécutable.

- Interprétation + Exécution : elle se fait en appelant l'interprète de byte-code de la machine, avec la commande `java` suivie du nom du programme. On voit s'afficher le message attendu :
  ```
  Java/Essais> java Bonjour
  Bonjour tout le monde !
  ```

L'exemple avec des erreurs

Nous changeons le code source afin d'y introduire quelques erreurs et voir les messages de diagnostique donnés par le compilateur. Le code est sauvegardé dans le fichier `Bonjour2.java` :

```
public classe Bonjour2 {
    public static void main (String[] args) {
        Terminal.ecrireStringln("Bonjour tout le monde !"*2);
    }
}
```

Ce code contient deux erreurs :
- *erreur de syntaxe* : le mot-clé `class` qui introduit le nom du programme a été changé en `classe`.
- *erreur de typage (sémantique)* : le message `"Bonjour tout le monde !"` est une chaîne de caractères (type `String`) et en tant que tel ne peut être multiplié par 2.

```
Java/Essais> javac Bonjour2.java
Bonjour2.java:1: 'class' or 'interface' expected
public classe Bonjour2 {
       ^
1 error
```

Le compilateur s'arrête à la première erreur trouvée (ligne 1). Nous corrigeons et relançons la compilation :

```
Java/Essais> javac Bonjour2.java
Bonjour2.java:3: operator * cannot be applied to java.lang.String,int
        Terminal.ecrireStringln("Bonjour tout le monde !"*2);
                                                            ^
1 error
```

Ce message est plus difficile à comprendre. Il nous dit que à la ligne 3, l'opérateur `*` n'est pas applicable sur le type `String`.

1.7 Analyse des problèmes

Voici une démarche posible pour développer un programme à partir de l'énoncé d'un problème :

1. *Déterminer le problème à résoudre :*
   (a) quelles sont les données d'entrées et leur nature ?
   (b) quelles sont les données attendues en sorties et leur nature ?
2. *Déterminer la méthode :*
   (a) comment modéliser les données (d'entrée, de sortie, intermédiaires) ?
   (b) quels sont les différents cas des entrées à traiter, et les cas écheant, quels sont les traitement associés.
   (c) exprimer sous-forme d'algorithme la manière d'obtenir le résultat final à partir des traitements sur les données d'entrée.
3. *Correction, complétude et lisibilite*
   (a) A-t-on bien prevu tous les cas des entrées et sorties ?
   (b) Obtient-on dans chaque cas ce que l'on voulait calculer ?
   (c) Notre algorithme est-il comprehensible par quelqu'un d'autre ?
4. *Tests*
   (a) Elaborer un jeu de tests representatif de tous les cas possibles (univers) des entrées. Un *jeu de tests* consiste en une suite de "valeurs typiques" pour chacune des entrées, accompagnées des valeurs attendus comme résultat dans ce cas. Par exemple, si on veut tester un programme qui calcule le carre d'un nombre entier, avec entrée x (entier) et sortie z (entier) ; un jeu de tests (non exhaustif) peut être $\{(x \leftarrow 0, z \mapsto 0); (x \leftarrow 1, z \mapsto 1); (x \leftarrow 3, z \mapsto 9)\}$
   (b) Confronter le fonctionnement de l'algorithme avec le jeu de tests proposé.

Les algorithmes

Un algorithme est un énoncé détaillé sous forme de pas à suivre dans l'application des traitements ou calculs sur les données. Il s'agit d'une *méthode systématique* de résolution d'un problème.

Exemple 1 : la feuille de déclaration d'impôts sur le revenu est accompagnée d'une description des opérations pour calculer le montant de l'impôt :

- le *problème* : le calcul du montant de votre impôt
- les *données* : les chiffres de vos salaires, charges, abattements, etc.
- *l'algorithme* : ou méthode de résolution du problème, c'est la suite des calcule à réaliser sur les données.
- le *résultat ou sortie* : le montant de votre impôt.

Il peut y avoir plusieurs méthodes pour résoudre un problème. Par exemple, on peut commencer par calculer le quotient familial, plutôt que par le calcul des abattements. □

Exemple 2 : Une recette de cuisine est un exemple classique d'algorithme.

Problème : Préparation d'une omelette

Données : Ce sont les ingredients,
- 2 oeufs,
- sel,
- un peu de matière grasse

Algorithme : C'est la méthode de préparation,

1. Casser les oeufs dans un bol,
2. Y ajouter du sel, puis les battre,
3. Faire chauffer la matière grasse dans une pôele,
4. Verser le mélange des oeufs dans la pôele et faire cuire doucement jusqu'à la consistance souhaitée.

□

En pratique, un problème est rarement aussi simple (à comprendre, à modéliser, à resoudre). Par exemple, considérez le problème :

Etant donné le réseau routier d'une ville et la situation de la circulation, calculer le plus court chemin pour aller d'une adresse à une autre.

Un exemple

Problème : Calculer et afficher la conversion en francs d'une somme en euros saisie au clavier.

Analyse :

1. Déterminer le problème à résoudre :
   (a) les entrées et leur nature ⇒ un nombre réel eu
   (b) les sorties attendues ⇒ un réel fr tel que $fr = eu * 6.559$

2. Déterminer la méthode :
   (a) modélisation des données : fr, eu modélisés par des nombres flottants (type `double` de Java).
   (b) l'algorithme

1. lire la valeur saisie pour eu,
2. calculer $fr = eu \times 6.559$
4. afficher le résultat final fr

(c) un jeu de tests :

$(eu \leftarrow 0.0, fr \mapsto 0.0)$;
$(eu \leftarrow 15.0, fr \mapsto 98.35)$;
$(eu \leftarrow -3.5, fr \mapsto -22.9565)$;
$(eu \leftarrow 10, fr \mapsto 65.59)$;

Résultats de l'analyse

Le résultat principal de l'analyse est l'algorithme, que l'on doit accompagner des données sur lequel il agit, afin de le rendre comprehensible. Il devient ainsi l'analogue d'une recette de cuisine : on donne la liste des ingredients, puis la procédure à suivre pour les mélanger.

Données + algorithme :

- *Données* :
  entrées : eu nombre réel
  sortie : fr nombre réel
- *Algorithme* :
  1. lire eu
  2. calculer $fr = eu \times 6.559$
  3. afficher le résultat se trouvant dans fr

Un codage en Java

Listing 1.2 – (lien vers le code brut)

```
1  /* Un premier programme:
2    - son nom est Conversion
3    - il est sauvegarde dans le fichier Conversion.java
4    - il est compile par javac Conversion.java
5    - il est execute par java  Conversion
6  */
7
8  public class Conversion {
9      public static void main (String[] args) {
10         // Variables du programme
11         double euros, francs;
12         // Message pour la saisie
13         Terminal.ecrireStringln("Entrez la somme en euros: ");
14         // Lecture dans la variable euros
15         euros = Terminal.lireDouble();
16         // Calcul de la conversion
17         francs = euros * 6.559;
18         // Affichage du resultat
```

```
19            Terminal.ecrireStringln("La somme convertie en francs: "+ francs);
20        }
21 }
```

Suite et fin de l'exemple

Le code Java est mis dans un fichier nommé `Conversion.java`. Nous lançons la compilation puis l'exécution :

```
Java/Essais> javac Conversion.java
Java/Essais> java Conversion
Entrez la somme en euros:
10
La somme convertie en francs: 65.59
```

<u>*Tests*</u> : Nous répetons plusieurs fois l'exécution avec différents valeurs pour les entrées :

```
Java/Essais> java Conversion
Entrez la somme en euros:
-3
La somme convertie en francs: -19.677
Java/Essais> java Conversion
Entrez la somme en euros:
15.5
La somme convertie en francs: 101.6645
```

1.8 Résumé du chapitre 1

Les *ordinateurs* sont composés principalement d'un *processeur* (CPU), d'une *mémoire centrale*, et de dispositifs *périphériques*. Le processeur est chargé d'exécuter les programmes. La mémoire centrale contient les instructions du programme et ses données. Elle est organisée comment une suite d'emplacements appelés *mots*, et munis chacun d'une adresse pour un accès direct. Les périphériques sont les dispositifs de communication de l'ordinateur : clavier, écran, enceintes, ports de communication, disques durs.

Un *programme* sert à décrire une méthode mécanique, autrement dit, applicable par une machine, de resolution d'un problème. Il traite *des données d'entrée* et calcule et produit des solutions, qu'on appele aussi ses *sorties*. Pour obtenir les sorties, une machine *exécute* le programme sur ses données d'entrée.

Un programme est exprimé sous forme de *suite d'instructions* dans un *langage de programmation*. Il existe deux grandes familles de langages : les *langages de haut niveau*. (ex : C, Ada, Pascal, Cobol, Java, OCaml, Python), et les *langages de bas niveau* ou *langages machine*, qui sont propres à chaque plateforme matérielle (SPARC/Sun, Intel/PC, etc). Les langages de haut niveau sont puissants et compréhensibles par les humains, mais ils ne sont pas directement exécutables par une machine. Ils sont à *traduire au préalable en langage machine*.

Un *langage de haut niveau* est décrit par ses types, sa syntaxe et sa sémantique. *Les types des données* servent à décrire les données du programme, p.e., le type `int` en Java sert à modéliser les nombres entiers. *La syntaxe* décrit les règles de *formation textuelle* des instructions et des programmes, p.e, pour écrire l'expression mathématique $1 \leq x \leq 7$, une syntaxe possible en Java est : `1 <= x && x <= 7`. *La sémantique* précise, d'une part le *comportement* ou le *sens* des contructions syntaxiques lorsqu'elles sont executées par une machine et d'autre part, les *contraintes de cohérence entre types*.

Un *algorithme* pour la résolution d'un problème est un énoncé détaillé sous forme de pas à suivre pour calculer les solutions ou sorties pour le problème à partir de ses données d'entrée.

La *production d'un programme* suit les phases suivantes :

1. *Analyse et Conception* : On précise le problème à resoudre et les données (*entrées* et *sorties*) du problème. Puis, on énonce une méthode de resolution sous forme d'algorithme.
2. *Codage* : traduction de l'algorithme en langage de programmation, et sous forme de fichier texte : c'est le *code source* du programme.
3. *Mise au point* : comprend la compilation, tests et correction d'erreurs.

Un *compilateur* est un traducteur de programmes : il traduit chaque instruction de haut niveau se trouvant dans le code source en plusieurs instructions machine ou de pseudo-code équivalentes. Le résultat de la compilation du code source est un nouveau fichier, nommé *code objet*, s'il est composé d'instructions machine (fichier binaire), ou pseudo-code, si c'est un fichier texte composé d'instructions en pseudo-code. Un compilateur réalise aussi de traitements *d'analyse syntaxique* et de *typage* du code source. Il ne réalise la traduction du code source que si aucune erreur n'est détectée lors de ces analyses.

Il existe une autre sorte de traducteur : les *interprètes*. Pour chaque instruction du programme, ils réalisent, l'une après l'autre, traduction en code machine, puis exécution. En Java, on compile vers un code intermédiaire ou *byte-code* identique pour tous les ordinateurs. Ensuite, un interprète propre à chaque machine, traduit le byte-code vers du code machine et l'exécute. L'ensemble des interprètes est c'est que l'on appelle la Machine Virtuelle Java (JVM).

Chapitre 2

L'essentiel pour commencer

L'objectif de ce chapitre est de vous donner les bases du langage Java vous permettant d'écrire vos propres programmes. Nous y passons donc rapidement en revue les éléments de base du langage Java, en vous expliquant *à quoi ils servent* et *comment ils s'écrivent*. Nous ne serons évidemment pas exhaustifs, c'est pourquoi les notions abordées dans ce chapitre feront l'objet soit de chapitres particuliers dans la suite du cours, soit d'annexes dans les notes.

Pour illustrer notre propos, nous allons reprendre l'exemple de la conversion d'une somme des euros vers les francs :

Problème : Calculer et afficher la conversion en francs d'une somme en euros saisie au clavier.

Listing 2.1 – (lien vers le code brut)

```
public class Conversion {
    public static void main (String[] args) {
        double euros, francs;
        Terminal.ecrireStringln("Somme en euros? ");
        euros = Terminal.lireDouble();
        francs = euros * 6.559;
        Terminal.ecrireStringln("La somme en francs: "+ francs);
    }
}
```

2.1 Structure générale d'un programme Java

Ce programme a un squelette identique à tout autre programme Java. Le voici :

Listing 2.2 – (pas de lien)

```
public class ... {
    public static void main (String[] args) {
      ....
    }
}
```

Autrement dit :

- Tout programme Java est composé au minimum d'une *classe* (mot-clé `class`) dite *principale*, qui elle même, contient une *méthode* de nom `main` (c'est aussi un mot-clé). Les notions de *classe* et de *méthode* seront abordées plus tard.
- **public, class, static, void, main** : sont des mots reservés c'est à dire qu'ils ont un sens particuliers pour Java. On ne peut donc pas les utiliser comme nom pour des classes, des variables, etc...
- La **classe principale** : celle qui contient la méthode `main`. Elle est déclarée par

  public class *Nom_classe*

  C'est vous qui choisissez le nom de la classe.
  Le code qui définit une classe est délimité par les caractères { et }
- **Nom du programme :** Le nom de la classe principale donne son nom au programme tout entier et doit être également celui du fichier contenant le programme, complété de l'extension `.java`
- La **méthode** `main` : obligatoire dans tout programme Java : c'est elle qui "commande" l'exécution. Définie par une suite de déclarations et d'actions délimitées par { et }. Pour l'instant, et jusqu'à ce que l'on sache définir des sous-programmes, c'est ici que vous écrirez vos programmes.

Autrement dit, pour écrire un programme que vous voulez appeler `"truc"`, ouvrez un fichier dans un éditeur de texte, appelez ce fichier `"truc.java"` ; Ecrivez dans ce fichier le squelette donné plus haut, puis remplissez les ... :

2.2 comprendre le code

Le programme Java exprime la chose suivante dans le langage Java (c'est-à-dire au moyen de phrases comprises par le compilateur Java) :
(ligne 4 :) déclarer 2 variables appelées `euros` et `francs` destinées à contenir des réels **puis**
(ligne 6 :) afficher à l'écran la phrase `Somme en euros? :` **puis**
(ligne 7 :) récupérer la valeur entrée au clavier et la stocker dans la variable `euros`, **puis**
(ligne 8 :) Multiplier la valeur de la variable `euros` par 6.559 et stocker le résultat de ce calcul dans la variable `francs`, **puis**
(ligne 9 :) afficher à l'écran la valeur de `francs`

La première chose importante à remarquer est que pour écrire un programme, on écrit une suite d'ordres séparés par des `;`. L'exécution du programme se fera en exécutant d'abord le premier ordre, puis le second, etc. C'est ce qu'on appelle le *principe de séquentialité de l'exécution*.

Dans ce programme on a trois catégories d'ordres :

- des déclarations de variables. Les déclarations servent à donner un nom à une case mémoire, de façon a pouvoir y stocker, le temps de l'exécution du programme, des valeurs. Une fois qu'une variable est déclarée et qu'elle possède une valeur, on peut consulter sa valeur.
- des instructions d'entrée-sortie. Un programme a bien souvent (mais pas toujours) besoin d'informations pour lui viennent de l'extérieur. Notre programme calcule la valeur en francs d'une somme en euros *qui est donnée au moment de l'exécution* par l'utilisateur. On a donc besoin de dire qu'il faut faire `entrer` dans le programme une donnée par le biais du clavier. Il y a en Java, comme dans tout autre langage de programmation, des ordres prédéfinis qui permettent de faire cela (aller chercher une valeur au clavier, dans un fichier, sortir du programme vers l'écran ou un fichier une valeur etc...).
- l'instruction d'affectation (`=`) qui permet de manipuler les variables déclarées. Elle permet de mettre la valeur de ce qui est a droite de `=` dans la variable nommée à gauche du `=`.

2.3 Les variables

Les variables sont utilisées pour stocker les données du programme. A chaque variable correspond un emplacement en *mémoire*, où la donnée est stockée, et un nom qui sert à désigner cette donnée tout au long du programme.
Une variable doit être déclarée dans le programme. On peut alors consulter sa valeur ou modifier sa valeur.

2.3.1 déclaration

Nous avons déjà vu une déclaration : `double euros, francs;`
Cette déclaration déclare 2 variables à la fois de nom respectif `euros` et `francs`. La forme la plus simple de déclaration de variables est la déclaration d'une seule variable à la fois. On aurait pu remplacer notre déclaration précédente par celles ci :

```
double euros;
double francs;
```

Ici, on déclare d'abord la variable `francs` puis la variable `euros`.

le nom des variables

`euros` ou `francs` sont les noms des variables et ont donc été choisis librement par l'auteur du programme. Il y a cependant quelques contraintes dans le choix des symboles constituant les noms de variables. Les voici :

- Les noms de variables sont des *identificateur* c'est à dire commencent nécessairement par une lettre, majuscule ou minuscule, qui peut être ou non suivie d'autant de caractères que l'on veut parmi l'ensemble a ...z, A ...Z, 0..9, _, $ *Unicode*.
- Un nom de variable ne peut pas être un mot reservé : (`abstract, boolean, if, public, class, private, static`, etc).
- Certains caractères ne peuvent pas apparaître dans un nom de variable (`^, [, ], {, +, -,` ...).

Exemples : `a`, `id_a` et `X1` sont des noms de variables valides, alors que `1X` et `X-X` ne le sont pas.

le type de la variable

Dans notre exemple de déclaration `double euros;`, le mot `double` est le nom d'un type prédéfini en Java. Un type est un ensemble de valeurs particulières connues de la machine. Nous allons pendant quelques temps travailler avec les types java suivants :

- Le type `int` désigne l'ensemble de tous les nombres entiers représentables en machine sur 32 bits (31 plus le signe) $\{-2^{31}, \dots, 2^{31}\}$. Les éléments de cet ensemble sont `0, 1, 2`...
- le type `double` désigne les réels (à précision double 64 bits). Les éléments de cet ensemble sont `0.0, 0.1, ...18.58 ...`
- Le type `boolean` modélise les deux valeurs de vérité dans la logique propositionnelle. Ses éléments sont `true` (vrai) et `false` (faux).
- Le type `char` modélise l'ensemble des caractères Unicode (sur 16 bits). Ses éléments sont des caractères entourés de guillemets simples, par exemple : `'a'`, `'2'`, `'@'`;

– le type `string` modélise les chaînes de caractères. Ses éléments sont les suites de caractères entourées de guillemets doubles : `"coucou"`, `"entrer une somme en euros :?"`, `"a"`...

syntaxe des déclarations de variables

Ainsi, pour déclarer une variable, il faut donner un nom de type parmi `int, double, char, boolean, string` suivi d'un nom de variable que vous inventez.

Pour déclarer plusieurs variables de même type en même temps, il faut donner un nom de type suivi des noms de vos variables (séparés par des virgules).

C'est ce qu'on appelle la *syntaxe* Java des déclarations de variables. Il faut absolument se conformer à cette règle pour déclarer des variables en Java, faute de quoi, votre code ne sera pas compris par le compilateur et produira une erreur. Ainsi, `int x; string "coucou"` sont corrects alors que `entier x;` ou encore `x int;` seront rejetés.

Execution des déclarations de variables

L'exécution d'un programme, rapellons le, consiste en l'exécution successive de chacune de ses instructions.

Que se passe-t-il lorsqu'une déclaration est rencontrée ? Une place mémoire libre de taille suffisante pour stocker une valeur du type de la variable est recherchée, puis le nom de la variable est associé a cette case mémoire. Ainsi, dans la suite du programme, le nom de la variable pourra être utilisé et désignera cet emplacement mémoire.

Une question un peu subtile se pose : combien de temps cette liaison entre le nom de la variable et la case mémoire est elle valable ? Autrement dit, jusqu'à quand la variable est-elle connue ? Pour l'instant, nous pouvons répondre simplement : cette liaison existe durant l'exécution de toutes les instructions qui suivent la déclaration de la variable et jusqu'à la fin du programme.

Cette notion (la portée des variables) se compliquera lorsque nous connaîtrons plus de choses en Java, notamment la notion de bloc. Nous y reviendrons à ce moment là.

2.3.2 Affecter une valeur à une variable

Une fois qu'une variable a été declarée, on peut lui donner une valeur, c'est à dire mettre une valeur dans la case mémoire qui lui est associée. Pour cela, il faut utiliser l'instruction d'affectation dont la syntaxe est

nom_variable = *expression* ;

par exemple `x=2;`

Il est de bonne habitude de donner une valeur initiale à une variable dès qu'elle est déclarée. Java nous permet d'affecter une valeur à une variable au moment de sa déclaration :
par exemple `int x=2;`

Des valeurs du bon type

A droite de =, on peut mettre n'importe quelle valeur *du bon type*, y compris des variables. Comme `2` est une valeur du type `int`, notre exemple ne sera possible que si plus haut dans notre programme,

on a la déclaration `int x;` [1].

Les expressions

Les valeurs ne sont pas forcément simples, elles peuvent être le résultat d'un calcul. On pourrait par exemple écrire `x=(18+20)*2;`. On peut écrire l'expression `(18+20)*2` car `+` et `*` sont des *opérateurs* connus dans le langage Java, désignant comme on s'y attend l'addition et la multiplication et que ces opérateurs, appliqués à des entiers, calculent un nouvel entier. En effet, $18 + 20$ donne 38 (un entier) et 38×2 donne 76 qui est un entier.

– Pour construire des expressions arithmétiques, c'est à dire des expressions dont le résultat est de type `int` ou `double` on peut utiliser les opérateurs :`+,-,*,/,%(reste de la division entière)`
– Pour construire des expressions booléennes (i.e dont le résultat est de type `boolean`, on peut utiliser les opérateurs : `&&`(et), `||` (ou), `!`(non).
– Les opérateurs de comparaison permettent de comparer deux *valeurs* ou *expressions* toutes les deux de de type numérique ou `char` et renvoient une valeur booléenne en résultat.
`==` (egalité), `<` (plus petit), `>` (plus grand), `>=` (plus grand ou égal), `<=` (plus petit ou égal), `!=` (différent).

Exécution d'une affectation

L'exécution d'une affectation se fait en deux temps :

1. On calcule le résultat de l'expression **à droite** de `=` (on dit évaluer l'expression). Si c'est une valeur simple, il n'y a rien a faire, si c'est une variable, on va chercher sa valeur, si c'est une expression avec des opérateurs, on applique les opérateurs à leurs arguments. Cela nous donne une valeur qui doit être du même type que la variable à gauche de `=`.
2. On met cette valeur dans l'emplacement mémoire associé à la variable **à gauche** de `=`.

C'est ce qui explique que l'on peut écrire le programme suivant :

Listing 2.3 – (lien vers le code brut)

```
public class essaiVariable {
    public static void main (String[] args) {
      int x;
      int y;
      y=2;
      x=y+5;
      x=x+2;
    }
}
```

Les lignes 3 et 4 déclarent les variables `x` et `y`. Elle sont donc connues dans la suite du programme. Puis (l5) la valeur 2 est donnée à `y`. Ensuite est calculé `y+5`. La valeur de `y` en ce point de l'exécution est `2` donc `y+5` vaut `7`. A l'issue de l'exécution de la ligne 6, `x` vaut `7`. Finalement, (l7), on évalue `x+2`. `x` à ce moment vaut `7` donc `x+2` vaut `9`. On donne à `x` la valeur `9`.

[1] Ceci n'est pas tout à fait exact. Nous apprendrons plus tard que nous pouvons affecter à une variable des valeurs d'autres types qui sont de la même famille (conversion implicite de type)

MCOURS.COM
Le N°1 du cours et exercices sur Internet

Cet exemple illustre le fait qu'une variable peut (puisque l'exécution d'un programme est séquentielle) avoir plusieurs valeurs successives (d'où son nom de variable), et que l'on peut faire apparaître une même variable à gauche et à droite d'une affectation : on met à jour son contenu en tenant compte de la valeur qu'elle contient juste avant.

2.4 Les appels de méthodes prédéfinies

Revenons à notre exemple de départ, la conversion en francs d'une somme en euros. Nous avions expliqué que les lignes 6 et 7 du programme effectuaient des opérations d'entrée sortie. Nous allons analyser cela plus en détail.

2.4.1 La méthode `Terminal.ecrireStringln`

La ligne 6 donne l'ordre d'afficher à l'écran le message `somme en euros? :`
Elle le fait en utilisant un programme tout fait que nous avons écrit pour vous. On appelle cela en Java une *méthode*.
Ce programme s'appelle `ecrireStringln`. Il fait partie de la classe `Terminal`.

A condition d'avoir copié le fichier `Terminal.java` contenant cette classe dans votre répertoire de travail, vous pourrez utiliser autant de fois que vous le désirez cette méthode. Utiliser une méthode existante dans un programme s'appelle *faire un appel* de la méthode.

Pour fonctionner, cette méthode a besoin que l'utilisateur, lorsqu'il l'utilise, lui transmette une information : la chaîne de caractère qu'il veut afficher à l'écran. Cette information transmise par l'utilisateur de la méthode est ce qui se trouve entre les parenthèses qui suivent le nom de la méthode. C'est ce qu'on appelle *l'argument* ou le *paramètre* de la méthode.
On a ici utilisé `Terminal.ecrireStringln` pour afficher le message `Somme en euros? :` en faisant l'appel `Terminal.ecrireStringln("Somme en euros? :);`
Pour afficher coucou, il faut faire l'appel `Terminal.ecrireStringln("coucou")`.

Lors d'un appel, vous devez nécessairement transmettre une et une seule chaîne de caractère de votre choix à `Terminal.ecrireStringln` : c'est l'auteur de la méthode qui a fixé le nombre, le type et l'ordre des arguments de cette méthode, lorsqu'il l'a écrit. Ainsi on ne pourra écrire `Terminal.ecrireStringln(` ni `Terminal.ecrireStringln("coucou","bidule")`. Ces 2 lignes provoqueront des erreurs de compilation.

2.4.2 La méthode `Terminal.lireInt`

La ligne 7 donne l'ordre de récupérer une valeur au clavier.

```
euros=Terminal.lireDouble();
```

Cette ligne est une affectation =. On trouve à droite du = un appel de méthode.

La méthode appelée s'appelle `lireDouble` et se trouve dans la classe `Terminal`.

Le fait qu'il n'y ait rien entre les parenthèses indique que cette méthode n'a pas besoin que l'utilisateur lui fournisse des informations. C'est une méthode sans arguments.

En revanche, le fait que cet appel se trouve à droite d'une affectation indique que le résultat de son exécution produit un résultat (celui qui sera mis dans la variable `euros`). En effet, ce résultat est la valeur provenant du clavier. C'est ce qu'on appelle la *valeur de retour* de la méthode.

Comme pour les arguments de la méthode, le fait que les méthodes retournent ou pas une valeur est fixé par l'auteur de la méthode une fois pour toutes. Il a fixé aussi le type de cette valeur de retour. Une méthode, lorsqu'elle retourne une valeur, retourne une valeur toujours du même type. Pour `lireDouble`, cette valeur est de type `Double`.

Lorsqu'ils retournent un résultat, les appels de méthode peuvent figurer dans des expressions.
Exemple : La méthode `Math.min` :prends 2 arguments de type int et retourne une valeur de type `int` : le plus petit de ses deux arguments. Ainsi l'instruction `x = 3 + (Math.min(4,10) + 2);` donne à `x` la valeur `9` car `3+(4+2)` vaut `9`.
L'instruction `x = 3 + (Terminal.lireInt() + 2);` a aussi un sens. Elle s'exécutera de la façon suivante : Pour calculer la valeur de droite, l'exécution se mettra en attente qu'une touche du clavier soit pressée (un curseur clignotant a l'écran indiquera cela). Dès que l'utilisateur presse une touche, le calcul de `Terminal.lireInt()` se termine avec pour résultat cette valeur. Imaginons que l'utilisateur ait pressé 6. 3+6+2 donne 11. La valeur de `x` sera donc 11.

Que se passera-t-il si l'utilisateur presse une touche ne correspondant pas à un entier ? Une erreur se produira, et l'exécution du programme tout entier sera arrêtée. Nous verrons plus tard qu'il y a un moyen de récupérer cette erreur pour relancer l'exécution.

2.4.3 Les appels de méthodes en général

De nombreuses classes contenant des méthodes existent en Java, soit dans la bibliothèque commune au langage, soit dans des répertoires particulier qu'il faut alors indiquer (nous verrons cela plus tard).

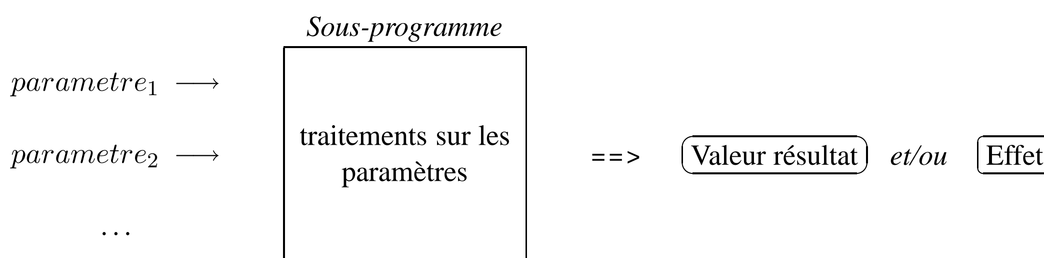
Toute méthode existante possède un nom, appartient à une classe, possède des arguments dont le nombre le type et l'ordre sont fixés. Une méthode peut renvoyer une valeur ; le type de la valeur de retour est alors fixé.

On trouve ces informations dans la documentation associée aux classes. Pour `Terminal.ecrireStringln` ces informations seraient données de la façon suivante :
`void ecrireStringln(String ch)` : `void` avant le nom indique que cette méthode ne renvoie pas de résultat, `(String ch)` indique qu'elle a un seul argument de type `String`.
`double lireDouble()` : `double` avant le nom indique que cette méthode renvoie un résultat de type `Double`, `()` indique qu'elle a n'a pas d'argument
Pour la méthode `min` de la classe `Math` on trouverait :
`int min(int a, int b)` :prends 2 arguments de type int et retourne une valeur de type `int` : le plus petit de ses deux arguments.

Pour appeler une méthode, il faut connaître toutes ces informations. l'appel de méthode sera conforme à la syntaxe suivante :

```
NomClasse.NomMethode(arg1,...,argn);
```

Ainsi, on peut voir un *sous-programme* comme une boîte noire capable de réaliser des traitements sur les *arguments* ou entrées du sous-programme. Si l'on désire effectuer les traitements, on *invoque* ou *appelle* le sous-programme en lui passant les données à traiter, qui peuvent varier pour chaque appel. En sortie on obtient : ou bien une valeur, résultat des calculs effectués, ou bien un *changement dans l'état de la machine* appelé aussi *effet* ; parfois, les deux.



2.5 Les méthodes prédéfinies d'entrée-sortie

Disons maintenant quelques mots sur les méthodes d'entrée-sortie.

Java est un langage qui privilégie la communication par interfaces graphiques. En particulier, la saisie des données est gérée plus facilement avec des fenêtres graphiques dédiées, des boutons, etc. Mais, pour débuter en programmation, il est beaucoup plus simple de programmer la saisie et l'affichage des données au terminal : la saisie à partir du clavier et l'affichage à l'écran. Nous vous proposons de suivre cette approche en utilisant la bibliothèque `Terminal`.

La bibliothèque `Terminal`

La classe `Terminal` (écrite par nous), regroupe les principales méthodes de saisie et d'affichage au terminal pour les types predéfinis que nous utiliserons dans ce cours : `int`, `double`, `boolean`, `char` et `String`. Le fichier source `Terminal.java`, doit se trouver présent dans le même répertoire que vos programmes. Pour l'employer, il suffit de faire appel à la méthode qui vous intéresse précédé du nom de la classe. Par exemple, `Terminal.lireInt()` renvoie le premier entier saisit au clavier.

Saisie avec `Terminal` : Se fait toujours par un appel de méthode de la forme :

`Terminal.lire`*`Type`*`();`

où *Type* est le nom du type que l'on souhaite saisir au clavier. La saisie se fait jusqu'à validation par un changement de ligne. Voici la saisie d'un `int` dans `x`, d'un `double` dans `y` et d'un `char` dans `c` :

```
int x; double y; char c; // Declarations
x = Terminal.lireInt();
y = Terminal.lireDouble();
c = Terminal.lireChar();
```

Méthodes de saisie dans `Terminal` :

- `Terminal.lireInt()`

- `Terminal.lireDouble()`
- `Terminal.lireChar()`
- `Terminal.lireBoolean()`
- `Terminal.lireString()`

Affichage avec `Terminal` : Les méthodes d'affichage traitent les mêmes types que celles de lecture. Il y a deux formats d'affichage :

- `Terminal.ecrireType(v);` affiche la valeur `v` qui doit être du type indiqué par le nom de la méthode.
- `Terminal.ecrireTypeln(v);` affiche comme avant plus un saut à la ligne.

```
Terminal.ecrireInt(5);
Terminal.ecrireInt('a');
Terminal.ecrireIntln(5);
Terminal.ecrireDoubleln(1.3);
```

Ce programme affiche :

```
5a5
5
1.3
```

L'affichage de messages, ou chaînes de caractères (type `String`), autorise l'adjonction d'autres messages ou valeurs en utilisant l'opérateur de *concaténation* `+`.

```
Terminal.ecrireString("bonjour " + 5 + 2 );   ---> affiche:  bonjour 52
Terminal.ecrireString("bonjour " + (5 + 2) ); ---> affiche:  bonjour 7
Terminal.ecrireString(5 + 2);   ---> Erreur de Typage!
```

Méthodes d'affichage :

- `Terminal.ecrireInt(n);`
- `Terminal.ecrireDouble(n);`
- `Terminal.ecrireBoolean(n);`
- `Terminal.ecrireChar(n);`
- `Terminal.ecrireString(n);`
- `Terminal.ecrireIntln(n);`
- `Terminal.ecrireDoubleln(n)`...

Affichage avec la bibliothèque `System`

La bibliothèque `System` propose les mêmes fonctionnalités d'affichage au terminal pour les types de base que notre bibliothèque `Terminal`. Elles sont simples d'utilisation et assez fréquentes dans les ouvrages et exemples que vous trouverez ailleurs. Nous les présentons brièvement à titre d'information.

- `System.out.print` : affiche une valeur de base ou un message qui lui est passé en paramètre.

  ```
  System.out.print(5);   ---> affiche 5
  System.out.print(bonjour);   ---> affiche le contenu de bonjour
  System.out.print("bonjour");   ---> affiche bonjour
  ```

  Lorsque l'argument passé est un message ou *chaîne de caractères*, on peut lui joindre une autre valeur ou message en utilisant l'opérateur de *concaténation* `+`. **Attention** si les opérandes de + sont exclusivement numériques, c'est leur addition qui est affichée !

```
System.out.print("bonjour " + 5 );    ---> affiche:  bonjour 5
System.out.print(5 + 2);    ---> affiche 7
```

– `System.out.println` : Même comportément qu'avant, mais avec passage à la ligne en fin d'affichage.

2.6 Conditionnelle

Nous voudrions maintenant écrire un programme qui, étant donné un prix Hors Taxe (HT) donné par l'utilisateur, calcule et affiche le prix correspondant TTC.
Il y a 2 taux possible de TVA : la TVA normale à 19.6% et le taux réduit à 5.5%. On va demander aussi à l'utilisateur la catégorie du taux qu'il faut appliquer.

données

– entrée : un prix HT de type `double` (`pHT`), un taux de type `int` (`t`) (0 pour normal et 1 pour réduit)
– sortie : un prix TTC de type `double` (`pTTC`)

algorithme

1. afficher un message demandant une somme HT à l'utilisateur.
2. recueillir la réponse dans `pHT`
3. afficher un message demandant le taux (0 ou 1).
4. recueillir la réponse dans `t`
5. 2 cas :
   (a) Cas 1 :le taux demandé est normal `pTTC= pht+ (pht*0.196)`
   (b) Cas 2 :le taux demandé est réduit `pTTC= pht+ (pht*0.05)`
6. afficher `pTTC`

Avec ce que nous connaissons déjà en Java, nous ne pouvons coder cet algorithme. La ligne 5 pose problème : elle dit qu'il faut dans un certain cas exécuter une tâche, et dans l'autre exécuter une autre tâche. Nous ne pouvons pas exprimer cela en Java pour l'instant, car nous ne savons qu'exécuter, les unes après les autres de façon inconditionnelle la suite d'instructions qui constitue notre programme.

Pour faire cela, il y a une instruction particulière en Java, comme dans tout autre langage de programmation : l'instruction conditionnelle, qui à la forme suivante :

```
if (condition) {instructions1}
else {instructions2}
```

et s'exécute de la façon suivante : si la condition est vraie c'est la suite d'instructions `instructions1` qui est exécutée ; si la condition est fausse, c'est la suite d'instructions `instructions2` qui est exécutée.

La condition doit être une expression booléenne c'est à dire une expression dont la valeur est soit `true` soit `false`.

Voilà le programme Java utilisant une conditionnelle qui calcule le prix TTC :

Listing 2.4 – (lien vers le code brut)

```
public class PrixTTC {
    public static void main (String[] args) {

        double pHT,pTTC;
        int t;
        Terminal.ecrireString("Entrer le prix HT: ");
        pHT = Terminal.lireDouble();

        Terminal.ecrireString("Entrer taux (normal->0 reduit ->1) ");
        t = Terminal.lireInt();
        if (t==0){
          pTTC=pHT + (pHT*0.196);
        }
        else {
          pTTC=pHT + (pHT*0.05);
        }
        Terminal.ecrireStringln("La somme TTC: "+ pTTC );
        }
}
```

Ce programme est constitué d'une suite de 8 instructions. Il s'exécutera en exécutant séquentiellement chacune de ces instructions :

1. déclaration de `pHT` et pTTC
2. déclaration de `t`
3. affichage du message `"Entrer le prix HT :"`
4. la variable `pHT` reçoit la valeur entrée au clavier.
5. affichage du message `"Entrer taux (normal->0 reduit ->1)"`
6. la variable `t` reçoit la valeur entrée au clavier (0 ou 1)
7. Java reconnaît le mot clé `if` et fait donc les choses suivantes :
   (a) il calcule l'expression qui est entre les parenthèses `t==0`. le résultat de `t==0` dépend de ce qu'a entré l'utilisateur. S'il a entré `0` le résultat sera `true`, sinon il sera `false`.
   (b) Si le résultat est `true`, les instructions entre les accolades sont exécutées. Ici, il n'y en a qu'une :`pTTC=pHT + (PHT*0.196);` qui a pour effet de donner a `pTTC` le prix TTC avec taux normal. Si le résultat est `false`, il exécution les instructions dans les accolades figurant après le `else` :`pTTC=pHT + (PHT*0.05);`
8. la valeur de `pTTC` est affichée. cette dernière instruction **est toujours exécutée** : elle n'est pas à l'intérieur des accolades du `if` ou du `else`. La conditionnelle a servi a mettre une valeur différente dans la variable `pTTC`, mais il faut dans les deux cas afficher cette valeur.

2.6.1 `if` sans `else`

Lorsqu'on veut dire : si `condition` est vraie alors faire ceci, sinon ne rien faire, on peut omettre le `else` ;

2.6.2 tests à la suite

Lorsque le problème à résoudre nécessite de distinguer plus de 2 cas, on peut utiliser la forme suivante :

```
if (condition1){s1}
else if (condition2) {s2}
...
else if (conditionp) {sp}
else {sf}
```

On peut mettre autant de `else if` que l'on veut, mais 0 ou 1 `else` (et toujours à la fin). `else if` signifie sinon si. Il en découle que les conditions sont évaluées dans l'ordre. La première qui est vraie donne lieu à la séquence d'instructions qui est y associée. Si aucune condition n'est vraie, c'est le `else` qui est exécuté.

2.7 Itération

Ecrivons un programme qui affiche à l'écran un rectangle formé de 5 lignes de 4 étoiles. Ceci est facile : il suffit de faire 5 appels consécutifs à la méthode `Terminal.ecrireStringln`

Listing 2.5 – (lien vers le code brut)

```
public class Rectangle {
    public static void main (String[] args) {
        Terminal.ecrireStringln("****");
        Terminal.ecrireStringln("****");
        Terminal.ecrireStringln("****");
        Terminal.ecrireStringln("****");
        Terminal.ecrireStringln("****");
    }
}
```

Ceci est possible, mais ce n'est pas très élégant ! Nous avons des instructions qui nous permettent de répéter des tâches. Elles s'appellent les instructions d'*itérations*.

2.7.1 La boucle for

La boucle for permet de répéter une tâche un nombre de fois connus à l'avance. Ceci est suffisant pour l'affichage de notre rectangle :

Listing 2.6 – (lien vers le code brut)

```
public class Rectangle {
    public static void main (String[] args) {
        for (int i=0;i<5;i=i+1){
          Terminal.ecrireStringln("****");
        }
    }
}
```

RÃ©pÃ¨te `i` fois, pour `i` allant de 0 à 4 les instructions qui sont dans les accolades, c'est à dire dans notre cas : afficher une ligne de 4 `*`.
Cela revient bien à dire : répète 5 fois "afficher une ligne de 4 étoiles".

Détaillons cela : La boucle `for` fonctionne avec un compteur du nombre de répétition qui est géré dans les 3 expressions entre les parenthèses.

- La première :`int i=0` donne un nom a ce compteur (`i`) et lui donne une valeur initiale `0`. Ce compteur ne sera connu qu'à l'intérieur de la boucle for. (il a été declaré dans la boucle for `int i`)
- La troisième : `i=i+1` dit que les valeurs successives de `i` seront 0 puis 0+1 puis 1+1, puis 2+1 etc.
- La seconde (`i<5`) dit quand s'arrête l'énumération des valeurs de `i` : la première fois que `i<5` est faux.

Grâce à ces 3 informations nous savons que `i` va prendre les valeurs successives 0, 1, 2, 3, 4. Pour chacune de ces valeurs successives, on répètera les instructions dans les accolades.

Jouons un peu

Pour être sur d'avoir compris, examinons les programmes suivants :

Listing 2.7 – (lien vers le code brut)

```
public class Rectangle {
    public static void main (String[] args) {
        for (int i=0;i<5;i=i+2){
          Terminal.ecrireStringln("****");
        }
    }
}
```

`i` va prendre les valeurs successives 0, 2, 4. Il y aura donc 3 répétitions. Ce programme affiche 3 lignes de 4 étoiles.

Listing 2.8 – (lien vers le code brut)

```
public class Rectangle {
    public static void main (String[] args) {
        for (int i=1;i<5;i=i+2){
          Terminal.ecrireStringln("****");
        }
    }
}
```

`i` va prendre les valeurs successives 1, 3. Il y aura donc 2 répétitions. Ce programme affiche 2 lignes de 4 étoiles.

Listing 2.9 – (lien vers le code brut)

```
public class Rectangle {
    public static void main (String[] args) {
        for (int i=1;i<=5;i=i+2){
          Terminal.ecrireStringln("****");
        }
    }
}
```

`i` va prendre les valeurs successives 1, 3, 5. Il y aura donc 3 répétitions. Ce programme affiche 3 lignes de 4 étoiles.

Listing 2.10 – (lien vers le code brut)

```
public class Rectangle {
    public static void main (String[] args) {
         for (int i=1;i==5;i=i+2){
           Terminal.ecrireStringln("****");
         }
    }
}
```

`i` ne prendra aucune valeur car sa première valeur (`1`) n'est pas égale à `5`. Les instructions entre accolades ne sont jamais exécutées. Ce programme n'affiche rien.

Les boucles sont nécessaires

Dans notre premier exemple, nous avons utilisé une boucle pour abréger notre code. Pour certains problèmes, l'utilisation des boucles est absolument nécessaire. Essayons par exemple d'écrire un programme qui affiche un rectangle d'étoiles dont la longueur est donnée par l'utilisateur (la largeur restera 4).
Ceci ne peut de faire sans boucle car le nombre de fois où il faut appeler l'instruction d'affichage dépend de la valeur donnée par l'utilisateur.
En revanche, cela s'écrit très bien avec une boucle `for` :

Listing 2.11 – (lien vers le code brut)

```
public class Rectangle2 {
  public static void main (String[] args) {
    int l;
    Terminal.ecrireString("combien de lignes d'etoiles ?:");
    l=Terminal.lireInt();
    for (int i=0;i<l;i=i+1){
           Terminal.ecrireStringln("****");
    }
  }
}
```

La variable `l` est déclarée dans la première instruction. Elle a une valeur à l'issue de la troisième instruction. On peut donc tout à fait consulter sa valeur dans la quatrième instruction (le `for`).
Si l'utilisateur entre 5 au clavier, il y aura 5 étapes et notre programme affichera 5 lignes de 4 étoiles.
Si l'utilisateur entre 8 au clavier, il y aura 8 étapes et notre programme affichera 8 lignes de 4 étoiles.

le compteur d'étapes peut intervenir dans la boucle

Le compteur d'étapes est connu dans la boucle. On peut tout à fait consulter son contenu dans la boucle.

Listing 2.12 – (lien vers le code brut)

```
public class Rectangle3 {
  public static void main (String[] args) {
    int l;
```

```
    Terminal.ecrireString("combien de lignes d'etoiles ?:");
    l=Terminal.lireInt();
    for (int i=1;i<=l;i=i+1){
         Terminal.ecrireInt(i);
      Terminal.ecrireStringln("****");
    }
  }
}
```

Ce programme affiche les lignes d'étoiles précédées du numéro de ligne.

C'est pour cela que l'on a parfois besoin que la valeur du compteur d'étapes ne soit pas son numéro dans l'ordre des étapes. Voici un programme qui affiche les n premiers entiers pairs avec n demandé à l'utilisateur.

Listing 2.13 – (lien vers le code brut)

```
public class Rectangle3 {
  public static void main (String[] args) {
    int n;
    Terminal.ecrireString("combien d'entiers pairs ?:");
    n=Terminal.lireInt();
    for (int i=0;i<n*2;i=i+2){
         Terminal.ecrireInt(i);
      Terminal.ecrireString(" , ");
    }
  }
}
```

Et voici un programme qui affiche les 10 premiers entiers en partant de 10 :

Listing 2.14 – (lien vers le code brut)

```
public class premEntiers {
    public static void main (String[] args) {
         for (int i=10;i>0;i=i-1){
           Terminal.ecrireString(i + ",");
         }
    }
}
```

Pour en savoir plus

La syntaxe de la boucle `for` en Java est beaucoup plus permissive que ce qui à été exposé ici. Nous nous sommes contentés de décrire son utilisation usuelle. Nous reviendrons plus en détail sur les boucles dans un chapitre ultérieur.

2.7.2 la boucle while

Certaines fois, la boucle `for` ne suffit pas. Ceci arrive lorsqu'au moment où on écrit la boucle, on ne peut pas déterminer le nombre d'étapes.

Reprenons notre exemple de calcul de prix TTC. Dans cet exemple, nous demandions à l'utilisateur d'entrer 0 pour que l'on calcule avec le taux normal et 1 pour le taux réduit.

Que se passe-t-il si l'utilisateur entre 4 par exemple ? 4 est diffèrent de 0 donc le `else` sera exécuté. Autrement dit : toute autre réponse que 0 est interprétée comme 1.
Ceci n'est pas très satisfaisant. Nous voulons maintenant améliorer notre programme pour qu'il redemande à l'utilisateur une réponse, tant que celle-ci n'est pas 0 ou 1.
Nous sentons bien qu'il faut une boucle, mais la boucle `for` est inadaptée : on ne peut dire a priori combien il y aura d'étapes, cela dépend de la vivacité de l'utilisateur ! Pour cela, nous avons la boucle `while` qui a la forme suivante :

```
while (condition) {
  instructions
}
```

Cette boucle signifie : tant que la condition est vraie, exécuter les instructions entre les accolades (le *corps de la boucle*)
Grâce à cette boucle, on peut répéter une tâche tant qu'un **évènement dans le corps de la boucle** ne s'est pas produit.
C'est exactement ce qu'il nous faut : nous devons répéter la demande du taux, tant que l'utilisateur n'a pas fourni une réponse correcte.

écriture de la boucle

La condition de notre boucle devra être une expression booléenne Java qui exprime le fait que la réponse de l'utilisateur est correcte. Pour faire cela, il suffit d'ajouter une variable à notre programme, que nous appellerons `testReponse`. Notre programme doit s'arranger pour qu'elle ait la valeur `true` dès que la dernière saisie de l'utilisateur est correcte, c'est à dire si `t` vaut 0 ou 1, et fausse sinon.

Notre boucle pourra s'écrire :

Listing 2.15 – (lien vers le code brut)

```
while (testReponse==false){
  Terminal.ecrireStringln("Entrer taux (normal->0 reduit ->1) ");
  t = Terminal.lireInt();
  if (t==0 || t==1){
    testReponse=true;
  }
  else {
    testReponse=false;
  }
}
```

Il faudra, bien entendu, avoir déclaré `testReponse` avant la boucle.

comportement de la boucle

La condition de la boucle est testée **avant** chaque exécution du corps de la boucle. On commence donc par tester la condition ; si elle est vraie le corps est exécuté une fois, puis on teste a nouveau la condition et ainsi de suite. L'exécution de la boucle se termine la première fois que la condition est fausse.

initialisation de la boucle

Puisque `testReponse==false` est la première chose exécutée lorsque la boucle est exécutée, il faut donc que `testReponse` ait une valeur **avant** l'entrée dans la boucle. C'est ce qu'on appelle *l'initialisation de la boucle*. ici, puisque l'on veut entrer au moins une fois dans la boucle, il faut initialiser `testReponse` avec `false`.

état de sortie de la boucle

Puisqu'on sort d'une boucle `while` la première fois que la condition est fausse, nous sommes sûrs que dans notre exemple, en sortie de boucle, nous avons dans `t` une réponse correcte : 0 ou 1. Les instructions qui suivent la boucle sont donc le calcul du prix TTC selon des 2 taux possibles.

Voici le code Java :

Listing 2.16 – (lien vers le code brut)

```java
public class PrixTTC2 {
  public static void main (String[] args) {

    double pHT,pTTC;
    int t=0;
    boolean testReponse=false;
    Terminal.ecrireString("Entrer le prix HT: ");
    pHT = Terminal.lireDouble();
    while (testReponse==false){
      Terminal.ecrireString("Entrer taux (normal->0 reduit ->1) ");
      t = Terminal.lireInt();
      if (t==0 || t==1){
        testReponse=true;
      }
      else {
        testReponse=false;
      }
    }
    if (t==0){
      pTTC=pHT + (pHT*0.196);
    }
    else {
      pTTC=pHT + (pHT*0.05);
    }
    Terminal.ecrireStringln("La somme TTC: "+ pTTC );
  }
}
```

Terminaison des boucles `while`

On peut écrire avec les boucles while des programmes qui ne s'arrêtent jamais. C'est presque le cas de notre exemple : si l'utilisateur n'entre jamais une bonne réponse, la boucle s'exécutera à l'infini.

Dans ce cours, les seuls programmes qui ne terminent pas toujours et que vous aurez le droit d'écrire

seront de cette catégorie : ceux qui contrôlent les saisies utilisateurs.

Pour qu'une boucle `while` termine, il faut s'assurer que le corps de la boucle contient des instructions qui mettent à jour la condition de boucle. Autrement dit, le corps de la boucle est toujours constitué de deux morceaux :
- le morceau décrivant la tâche à effectuer à chaque étape
- le morceau décrivant la mise à jour de la condition de sortie

Pour qu'une boucle termine **toujours**, il faut que **chaque** mise à jour de la condition de sortie, nous rapproche du moment où la condition sera fausse.

C'est bien le cas dans l'exemple suivant :

Listing 2.17 – (lien vers le code brut)

```
public class b1 {
    public static void main (String[] args) {

        int i=1;

        while (i!=10){
          Terminal.ecrireString(i + ",");
          i=i+1;
        }
    }
}
```

Au début de la première exécution du corps de boucle `i` vaut 1, Au début de la deuxième exécution du corps de boucle `i` vaut 2, ... A chaque fois, on progresse vers le moment où `i` vaut 10. Ce cas est atteint au bout de 10 étapes.

Attention, il ne suffit de se rapprocher du moment où la condition est fausse, il faut l'atteindre un jour, ne pas la rater.

Le programme suivant, par exemple, ne termine jamais :

Listing 2.18 – (lien vers le code brut)

```
public class b3 {
    public static void main (String[] args) {

        int i=1;

        while (i!=10){
          Terminal.ecrireString(i + ",");
          i=i+2;
        }
    }
}
```

Car `i` progresse de 1 à 3, puis à 5, à 7, à 9, à 11 et 10 n'est jamais atteint.

la boucle `while` suffit

La boucle `while` est plus générale que la boucle `for`. On peut traduire tous les programmes avec des boucles `for` en des programmes avec des boucles `while`. On peut donc se passer de la boucle

`for` ; il est cependant judicieux d'employer la boucle `for` à chaque fois que c'est possible, parce qu'elle est plus facile à écrire, à comprendre et à maîtriser.

Chapitre 3

Exécution des programmes

Le *sens* d'un programme est son comportement lorsqu'il est exécuté. Un programme est composé d'une suite d'instructions, qui sont exécutées les unes après les autres, *séquentiellement*. Chaque instruction peut effectuer trois sortes d'actions : d'entrée ou de sortie des données, de changement sur les variables, c'est à dire sur les cases mémoires qui leur sont associées. Ainsi chaque instruction modifie l'état mémoire de la machine, et un programme est une suite de modifications successives de l'état mémoire de la machine. Pour être bref, l'exécution d'un programme est une suite d'états mémoires.

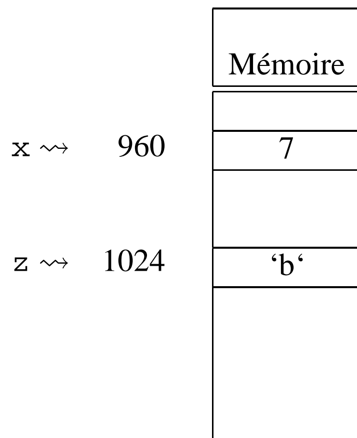
Comprendre un programme, c'est savoir comment il s'exécute et donc savoir retracer pas à pas cette suite d'états mémoires consécutifs. Cette suite d'états est connue aussi sous le nom de *trace d'une exécution*.

Nous allons étudier ces notions en détail dans ce chapitre.

3.1 Variables et mémoire

Les variables sont utilisées pour stocker les données du programme. A chaque variable correspond un emplacement en *mémoire*, où la donnée est stockée, et un nom qui sert à désigner cette donnée, ou ses valeurs futures, tout au long du programme.

La mémoire est divisée en emplacements, appelés *mots*, possèdant chacun une *adresse* qui permet d'acceder au contenu du mot de manière directe. Selon la nature de la valeur contenue dans une variable, elle peut occuper un ou plusieurs mots de suite. Ainsi, chaque variable correspond à son adresse de stockage en mémoire, et par voie de conséquence, à la valeur qui s'y trouve. Nous représentons la mémoire graphiquement :



Dans cet exemple, la variable `x` de type `int` est stockée à l'adresse 960, et contient la valeur 7, alors qu'à la variable `z`, de type `char`, correspond l'adresse 1024, dont le contenu est le caractère `'b'`. Le programme peut ensuite consulter (*lire*) le contenu d'une variable ou le modifier en *écrivant* une nouvelle valeur dans l'emplacement associé.

État de la mémoire

Nous désignons par ce terme, l'ensemble des valeurs stockées en mémoire, pour les variables du programme, à un moment donné de l'exécution. Il s'agit d'une sorte de photo de la mémoire du programme. Le dessin plus haut est un exemple, où la variable `x` vaut 7, et `z` vaut `'b'`. Nous l'appellerons également *Environnement du programme*.

Consulter la valeur d'une variable

Quel est le sens à donner à une variable si elle apparaît dans une action où sa valeur est nécessaire ? Par exemple, quel est le sens de `x` dans `x+2` ? La valeur d'une variable dépend de l'état courant de la mémoire.

- `System.out.println(x)` affiche non pas la lettre `x`, mais le contenu de la variable `x` en mémoire, au moment où cet instruction est exécutée.
- L'expression `x + 2` s'évalue dans la valeur courante de `x` augmentée de 2.

Exercices : Considérez le code suivant :

```
y = x + 3;
Terminal.ecrireInt(z);
Terminal.lireDouble(w);
m = 5;
n = n + 2;
```

1. Dans ces actions, quelles sont les variables ayant besoin d'intitialisation préalable, et quelles sont celles qu'on peut ne pas initialiser ?
2. Donnez les déclarations et initialisations nécessaires à la compilation de ces actions.

3.2 Comportement des déclarations de variables

Le sens d'une déclaration de variable est de *changer l'état de la mémoire* par :

- *Création de l'Environnement du programme* (voir notion introduite plus haut).
- *Allocation* d'un emplacement libre en mémoire, de taille suffisante pour stocker une valeur du type déclaré. Les valeurs succésivement acquises par la variable seront stockées dans cet emplacement. **Attention** : Un mot de la mémoire est une suite de bits, chacun est nécessairement 0 ou 1. Il est donc impossible qu'un emplacement, et a fortiori une variable, ne contienne "rien". Par exemple, une fois la place pour une variable entière reservée, il y a dans cet emplacement un entier (qu'on ne connaît pas forcément). Par ailleurs, chaque emplacement ne permet de stocker qu'une valeur à la fois. Ainsi, une nouvelle valeur sera réécrite sur la valeur précédente, à la place de celle-ci.
- *Liaison* (ou association) de la valeur initiale lorsqu'elle est donnée. L'association entre une variable et sa valeur est appelé *liaison*. Ainsi, si un moment donné, la valeur 5 est stockée dans la variable `x`, on dira que `x` *est liée* à la valeur 5. Si une variable est déclarée sans valeur initiale, on dira que sa valeur est inconnue au moment de la déclaration (mais ce n'est pas rien !). La valeur inconnue est notée `?`

Exemple : Considérons les déclarations :

```
int x,y;
int a = 0;
int b = 0;
int z = 0;
```

L'état de la mémoire ou environnement du programme, après déclarations, est donné par :

x	?
y	?
a	0
b	0
z	0

rôle du type de la variable : Le fait de donner un type à une variable lors de sa déclaration permet de prévoir combien d'espace mémoire il faut réserver pour la variable. Par exemple, un `double` est plus gros a stocker qu'un `int`.

Exercices : Quel est l'environnement créé (mémoire) après les déclarations suivantes ?

```
char b = 'c';
char c = b;
boolean d;
int x,y = 3;
double m = 3.5;
```

3.3 Comportement d'une affectation

C'est une instruction qui modifie la valeur d'une *variable* en mémoire [1].
La syntaxe d'une affectation a la forme :

nom_variable = *expression* ;

– *Comportement* : Modifie en mémoire le contenu de la variable à gauche, avec la valeur qui résulte du calcul de l'expression à droite du symbole =

– *Exécution* : suit les pas suivants. Prenons l'exemple de l'exécution de `x = x+2` dans un état mémoire où `x` vaut 5 .

1. *évaluation de l'expression à droite dans l'environnement courant*
   évaluons `x+2` avec `x` vaut `5` $\Rightarrow$ 5+2 $\Rightarrow$ 7,
2. *Modification de la variable* en mémoire : `x` vaut 7,

Trace d'une exécution

Nous allons maintenant suivre pas à pas l'execution d'un programme comportant des declarations et des affectations. Pour y parvenir, nous allons faire la *trace de l'exécution* de ce programme. Nous donnons l'état de la machine après exécution de chacune des instructions du programme, autrement dit, l'état des entrées, l'état des sorties, et de la mémoire (les valeurs des variables du programme). Comme notre programme ne fait pas de sd'entrées ni de sorties, la dernière case sera vide.

Listing 3.1 – (lien vers le code brut)

```
1 public class T1 {
2     public static void main (String[] args) {
3         int x;
4         int y=2;
5         x=3+y;
6         x=x+1;
7         y=x*2;
8     }
9 }
```

Après l'instruction	Mémoire		Entrées	Sorties
	x	y		
Déclarations	?	2		
6	5	2		
7	6	2		
8	6	12		

[1] En java, l'affectation retourne aussi un résultat. Dans ce cours, ceci ne nous sera pas utile.

3.4 Durée de vie d'une variable

Lorsqu'on déclare une variable, on associe un nom (le nom de la variable) à une case mémoire. On peut ensuite désigner cette case mémoire en utilisant le nom de la variable. Cette association entre un nom et une case mémoire a une durée de vie limitée :

- La **création de l'association** entre le nom d'une variable et la case mémoire se fait lors de l'exécution de la déclaration.
- la **destruction de l'association** se fait lorsque la dernière instruction du *bloc* contenant la déclaration est exécutée.

3.4.1 La notion de bloc

Qu'est ce qu'un bloc ?
En Java, ils sont faciles a reconnaitre : **un bloc est une suite d'instructions contenue entre des accolades**. Ainsi, le corps d'une boucle est un bloc, les instructions entre les accolades d'un `if`, d'un `else`, d'un `elseif` sont des blocs. Les instructions entre les accolades du `main` forment aussi un bloc.

Un bloc est un environnement local de déclarations de variable. Ceci signifie que toute declaration n'est connue qu'à l'intérieur du bloc ou elle est déclarée.

Dans tous les programmes que nous vous avons présentés jusqu'ici, les variables étaient déclarées dans le plus gros bloc possible : celui du `main`. Leur vie se termine lorsque l'accolade fermante du `main` est exécutée, donc à la fin du programme.
Mais si une variable est déclarée dans un bloc plus interne (dans le corps d'une boucle par exemple) sa vie se termine lorsque l'accolade fermante de ce bloc { est exécutée.
Prenons un exemple

Listing 3.2 – (lien vers le code brut)

```
1      public static void main (String[] args) {/*debut bloc 1*/
2           int a=2;
3           Terminal.ecrireStringln("valeur de a: "+ a );
4           if (a==0){
5               /* debut bloc 2 */
6               int b=3+a;
7               Terminal.ecrireStringln("valeur de b: "+ b );
8           }       /* fin bloc 2*/
9           else {   /* debut bloc 3*/
10              int c=3+a;
11              Terminal.ecrireStringln("valeur de c: "+ c);
12          }       /* fin bloc 3*/
13         Terminal.ecrireStringln("valeur de a: "+ a );
14     }        /* fin bloc 1*/
15  }
```

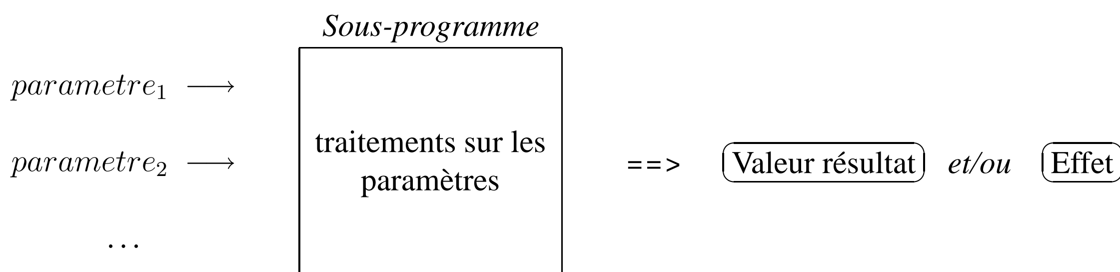
On a 3 blocs. Les blocs 2 et 3 sont à l'intérieur du bloc 1.
`a` est connue entre les lignes 4 et 14. Elle est donc connue aussi dans les bloc 2 et 3, puisqu'ils sont à l'intérieur du bloc 1.
`b` est connue entre les lignes 7 et 9.
`c` est connue entre les lignes 11 et 12.

3.5 Comportement d'un appel de méthode

La syntaxe d'un appel de sous-programme est de la forme :

$$\textit{Nom_sous_programme} \; (\; arg_1, arg_2, \ldots, arg_n \;)$$

Schématiquement, un *sous-programme* est une boîte noire capable de réaliser des traitents sur les *arguments* ou entrées du sous-programmes. Si l'on désire effectuer les traitements, on *invoque* ou *appelle* le sous-programme en lui passant les données à traiter, qui peuvent varier pour chaque appel. En sortie on obtient : ou bien une valeur, résultat des calculs effectués, ou bien un *changement dans l'état de la machine* appelé aussi *effet* ; parfois, les deux.



Exemples :

- `Math.min(3,8)` : appel de la fonction `Math.min` pour calculer le plus petit de ses deux arguments. Renvoie `3` en résultat.
- `Terminal.ecrireStringln("Bonjour")` : appel de la fonction `Terminal.ecrireStringln` du fichier `Terminal.java`. Provoque un *effet* de sortie avec l'affichage de `Bonjour`. Ne renvoie pas de valeur résultat.

L'appel d'un sous-programme se traduit *grosso modo* en plusieurs actions :

1. L'exécution séquentielle du programme s'arrête au point de l'appel et attend la fin du sous-programme pour reprendre.
2. Les arguments passés à l'appel sont transmis au sous-programme en tant qu'entrées.
3. Les traitements du sous-programme s'appliquent sur ces entrées,
4. S'il y a une valeur résultat elle est *renvoyée* au point de l'appel.

5. L'exécution du programme reprend au point d'arrêt. Le cas écheant, on recupère la valeur renvoyée pour la suite des calculs.

Considérons le morceau de programme :

```
x = 5;  // x recoit la valeur 5
x = 3 + Math.min(x,10) + 2;
        ^            ^
        |            |
point d'appel    point de retour avec resultat 5
```

- *1ère instruction* : on donne à `x` la valeur `5`,
- *2ème instruction* : avant d'évaluer l'expression `3 + Math.min(x,10) + 2`, on doit traiter l'appel de fonction :
  1. On suspend le calcul de l'expression,
  2. On appelle la fonction `Math.min` :
     - on lui *transmet* `5,10` en arguments,
     - le *contrôle du programme* est donné aux actions dans la fonction, qui calcule 5 ,
     - le *contrôle du programme* revient au point où il avait été suspendu, en retournant la valeur résultat 5,
  3. On reprend l'évaluation de l'expression avec la valeur retournée :
     `3 + Math.min(x,10) + 2` ⇒ `3 + 5 + 2` ⇒ `10`.

  On termine l'exécution de la 2ème instruction en modifiant la valeur de `x <- 10`.

Trace d'une exécution

Nous décrivons le comportement d'une exécution pour notre exemple. Nous donnons l'état de la machine après exécution de chacune des instructions du programme, autrement dit, l'état des entrées, l'état des sorties, et de la mémoire (les valeurs des variables du programme). Comme notre programme saisit des valeurs au clavier, nous supposons que des valeurs arbitraires sont effectivement tapées. Afin de rendre plus facile la description, nous ajoutons des numéros de ligne au programme étudié :

Listing 3.3 – (lien vers le code brut)

```
1  public class Conversion {
2      public static void main (String[] args) {
3
4          double euros, francs;
5
6          Terminal.ecrireStringln("Somme en euros? ");
7          euros = Terminal.lireDouble();
8          francs = euros * 6.559;
9          Terminal.ecrireStringln("La somme en francs: "+ francs);
10 }
```

Après l'instruction	Mémoire		Entrées	Sorties
	euros	francs		
Déclarations	?	?		
6	?	?		Somme en euros ?
7	10.0	?	10	
8	10.0	65.59		
9	10.0	65.59		La somme en francs : 65.59

Évolution de la mémoire :

euros	~~?~~	10.0
francs	~~?~~	65.59

3.6 Comportement du `if`

Rapellons l'essentiel sur cette instruction :

```
if  (cond) {                          ''si cond est vraie, faire suite1
    suite1
} else {                               sinon, faire suite2 ''
    suite2
}
```

Comportément : Selon la valeur de la condition, on réalise les actions de l'une des deux suites d'instructions `suite1` ou `suite2`. La séquence `suite1` est exécutée si `cond` est vraie, la séquence `suite2` est exécutée si `cond` est fausse. A chaque fois, exactement une des deux séquences est exécutée.

Listing 3.4 – (lien vers le code brut)

```
1  public class PrixTTC {
2      public static void main (String[] args) {
3
4          double pHT,pTTC;
5          int t;
6          Terminal.ecrireStringln("Entrer le prix HT: ");
7          pHT = Terminal.lireDouble();
8
9          Terminal.ecrireStringln("Entrer taux(0,1) : ");
10         t = Terminal.lireInt();
11         if (t==0){
```

```
12              pTTC=pHT + (pHT*0.196);
13            }
14            else {
15              pTTC=pHT + (pHT*0.05);
16            }
17            Terminal.ecrireStringln("La somme TTC: "+ pTTC );
18            }
19  }
```

Trace d'une exécution

Voici la trace d'exécution de ce programme :

Après l'instruction	Mémoire			Entrèes	Sorties
	pHt	t	pTTC		
Déclarations	?	?	?		
6	?	?			`Entrer le prix HT :`
7	10.0	?		10.0	
9	?	?			`Entrer le taux(0,1) :`
10	10.0	1		1	
11	10.0	1			
14	10.0	1			
15	10.0	1	10.5		
17	10.0	1	10.5		`La somme TTC : 10.5`

3.7 Comportement des boucles

Une boucle `while` en Java a la forme :

Listing 3.5 – (lien vers le code brut)

```
1    while (c) {                "tant que c est vrai, faire suiteInstructions"
2        suiteInstructions
3    }
```

où c est une expression booléenne *d'entrée* dans la boucle. Le comportement de cette boucle est :

1. c est évalué avant chaque itération.
2. Si c est vrai, on exécute $suiteInstructions$, puis le contrôle revient au point du test d'entrée (point 1).
3. Si c est faux, le contrôle du programme passe à l'instruction immédiatement après la boucle.

Exemple 1 :

Ce programme calcule la somme d'une suite de nombres entiers saisis au clavier. Le calcul s'arrête lors de la saisie du nombre zéro.

Listing 3.6 – (lien vers le code brut)

```
1  public class Somme {
2      public static void main (String[] args) {
3          int n, total;
4          // Initialisation de n, total
5          Terminal.ecrireString("Entrez un entier (fin avec 0): ");
6          n = Terminal.lireInt();
7          total = 0;
8          while ( n !=0 ) {
9              total = total + n;            // Calcul
10             Terminal.ecrireString("Entrez un entier (fin avec 0): ");
11             n = Terminal.lireInt();       // Modification variable du test
12         }
13         Terminal.ecrireStringln("La somme totale est: " + total);
14     }
15 }
```

`total` est *initialisée* à zéro, qui est l'élément neutre de l'addition; et `n` est initialisée avec première saisie. A chaque tour de boucle, une nouvelle valeur pour `total` est calculée : c'est la somme de la dernière valeur de `total` et du dernier nombre `n` saisi (lors de l'itération précédente). De même, une nouvelle valeur pour `n` est saisie. □

Remarque : Notez que cette boucle peut ne jamais exécuter $suiteInstructions$, si avant la toute première itération, le test c est faux. Dans l'exemple 1, c'est le cas, si lors de la première saisie, n vaut 0.

Trace de `Somme.java`

Étudions la trace d'une exécution de `Somme.java` en supposant saisis 5, 4, 7 et 0. Le tableau suivant montre l'évolution des variables pendant l'exécution. La toute première colonne donne la condition d'entrée à la boucle et les noms des variables modifiées. La colonne *init* donne les valeurs des variables avant la première itération. Chaque colonne *itération* k donne les renseignements suivants pour l'itération k :

- *pour le test* `(n!=0)` : sa valeur avant l'entrée à l'itération k;
- *pour les variables de la boucle* `total` et `n` : leurs valeurs en fin d'itération k.

	init	*itération* 1	*itération* 2	*itération* 3	*itération* 4
`(n!=0)`		`(5!= 0)`	`(4!= 0)`	`(7!= 0)`	`(0!= 0)`
`total`	0	0+5	0+5+4	0+5+4+7	*arrêt*
`n`	5	4	7	0	*arrêt*

Par exemple, le premier test réalisé est `(5 != 0)`, et après l'itération 1, `total` vaut 5 et `n` vaut 4. Lors de la dernière itération, le nombre 0 est saisit dans `n`, et `total` accumule la somme des valeurs précédentes de `n` (0+5+4+7). Au prochain test, la condition `(0 != 0)` est fausse et la boucle s'arrête. Le programme affiche la valeur 16 pour `total`. Voici les messages affichés par cette exécution :

```
% java Somme
Entrez un entier (fin avec 0): 5
Entrez un entier (fin avec 0): 4
Entrez un entier (fin avec 0): 7
Entrez un entier (fin avec 0): 0
La somme totale est: 16
```

Chapitre 4

Types, expressions et typage

Dans ce chapitre, nous allons revenir un peu plus en détail sur les notions de type et d'expression.

Un type désigne un ensemble de valeurs. Par exemple, le type `int` de Java, désigne l'ensemble de tous les nombre entiers représentables en machine sur 32 bits $\{-2^{31}, \ldots, 2^{31}\}$. En programmation, les types des données sont utilisés pour regrouper les valeurs qu'on veut manipuler en catégories bien distinctes. On cherche à distinguer les données selon leur nature (entiers, réels, chaînes de caractères, etc), dans une classification qui est sémantique. Différencier les valeurs selon leur type a au moins deux grands avantages :

- *Connaissance de la place nécessaire au stockage* : toutes les données ne sont pas représentées de la même manière. Pour un entier, on utilise en général un emplacement de 32 bits, alors que pour un boolean, un seul bit suffit. Dès lors, si on déclare le type d'une variable, le compilateur sait précisement combien de place il faudra reserver en mémoire pour la stocker.
- *Détection d'erreurs (typage)* : si un opérateur devant s'appliquer sur deux entiers, s'applique sur autre chose, le compilateur peut détecter cet erreur et la signaler au programmeur. Considérons :

  ```
  int x; boolean b = true;
  ```

  Le compilateur pourra détecter que l'instruction `x = 2 + b;` n'a pas de sens.

Il existe deux familles de types :

- **Types de base** : appelés aussi types simples. Ils modélisent les valeurs atomiques telles qu'un entier, un caractère, etc. Ils sont souvent prédéfinis, et en Java on les nomme *types primitifs*.
- **Types composés**. Ils modélisent les agrégats de plusieurs valeurs. En Java, on les appelle aussi *types références*, et certains sont prédéfinis (tableaux, chaînes de caractères).

Un type est décrit par :

- Ses *littéraux* ou *valeurs constantes*.
- Ses *opérateurs* permettant les opérations sur les valeurs du type.

4.1 Types de base (primitifs)

Il s'agit de `byte`, `short`, `int`, `long`, `char`, `float`, `double`, `boolean`. Nous commençons par une description des types et de leurs constantes et nous verrons plus tard leurs opérateurs.

4.1.1 Types numériques

Ces types modélisent les nombres entiers et flottants avec plusieurs tailles de représentation. Plus grande est la taille des mots utilisés en mémoire pour les répresenter, plus grande est leur précision.

1. Nombres entiers :
   – `byte` : entiers sur 8 bits.
   – `short` : entiers sur 16 bits.
   – `int` : entiers sur 32 bits.
   – `long` : entiers sur 64 bits.
   – Exemples de constantes : `-1276, -2, 0, 2L, 3`. L'utilisation du suffixe `L` permet de stocker la valeur sur un format `long` plutôt que `int`.
2. Nombres réels
   – `float` : flotants à précision simple (32 bits).
   – `double` : flottants à précision double (64 bits).
   – Exemples de constantes : `37.266`, `37.266d`, `37.266f`. Le suffixe `f` ou `F` indique le format `float`, alors que `d` ou `D` indique le format `double`. La notation `23.45e3` correspond à la valeur 23.45×10^3.

Remarque : Parmi les types numériques, nous n'utiliserons dans ce cours que `int` et `double`.

4.1.2 Les booléans

Le type `boolean` modélise les deux valeurs de vérité dans la logique propositionnelle. Ses constantes : `true` et `false`.

4.1.3 Les caractères

Le type `char` modélise l'ensemble des caractères Unicode (sur 16 bits). Les constantes sont, soit des caractères entourés de guillements simples, p.e : `'a'`, `'2'`, `'@'` ; soit l'encodage alphanumérique d'un caractère Unicode. Exemples :
– Nouvelle ligne : `\n`
– Tabulation : `\t`
– Saut de page : `\f`
– Backlash, apostrophe, guillemets : `\\ \' \"`
– Carcatère Unicode `0xFFFF` : `\uFFFF`

4.2 Opérateurs et expressions

Un *opérateur* permet de réaliser des calculs sur une ou plusieurs valeurs appelées *opérandes*. Les opérateurs agissant sur une opérande sont dits *unaires*, sur deux, *binaires* et sur trois, *ternaires*. Ex : `+` est un opérateur binaire, alors que l'opérateur de négation `!` est unaire. On écrira `3+x` et `!(true)`

Les *expressions* sont les constructions décrivant les opérations avec opérandes et opérateurs.

Les *opérandes* d'une expression peuvent être des constantes, des variables avec une valeur connue (ayant été initialisées) et également les appels de fonctions.
– Expression arithmétique `(2 + x * 5)` : `+` et `*` sont les *opérateurs* et `2,x,5` sont les opérandes.
– dans `(5 + Math.min(2,3))`, `Math.min` est le nom d'une fonction, et `2,3` sont les valeur ou *arguments* sur lesquelles on l'applique. Dans le cas présent, elle donne en résultat le nombre minimum parmi ses deux arguments, à savoir, 2.
– `2 > 5` est une expression *boléenne*. Son résultat est un booléan (`false`).

– `!(2 > 5)` est une expression *boléenne* lue “non 2 plus grand que 5”.

4.2.1 Résultat d’une expression

Une expression correspond toujours à la valeur du calcul qu’elle décrit. On dit qu’elle *renvoie* cette valeur.

Le *résultat* d’une expression est une valeur du langage. Selon qu’il soit numérique, booléan, etc, on parle d’expressions arithmétiques, booléennes, etc. *Ex* : l’expression `2+3*5` *renvoie* la valeur `17`. Nous utiliserons la notation `2+3*5` ⇒ `17`.

Exécuter une expression consiste à calculer sa valeur : *à l’évaluer*. Cette évaluation peut dépendre de la valeur des variables dans l’environnement du programme (mémoire).

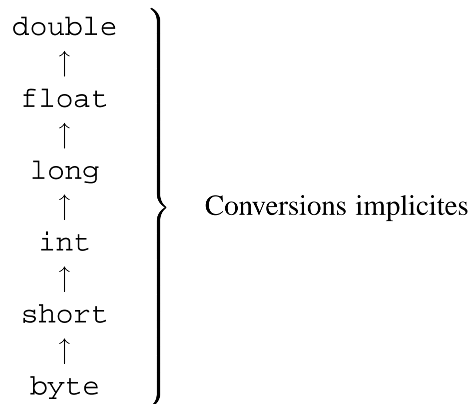
Exemples d’évaluation :
- Expressions simples :
  - `3` vaut ⇒ 3
  - `3+4` vaut ⇒ 7
- expression composée : `3+(2*7)`
  - on calcule `(2*7)` ⇒ 14,
  - puis, `3 + 14` ⇒ 17
- expression avec appels de fonctions : `3 + Math.min(2,5)`
  - on calcule `Math.min(2,5)` ⇒ 2,
  - puis, `3 + 2` ⇒ 5
- expression avec variables `7 + x` où `x=2` ⇒ `9`

4.2.2 Opérateurs arithmétiques

Ils permettent d’opérer sur toutes les valeurs des types numériques de Java : `byte`, `short`, `int`, `long`, `float` et `double`.
`+` (addition),
`-` (soustraction),
`*` (multiplication),
`/` (division entière).
`%` (reste de la division entière).

4.2.3 Conversions implicites entre types (primitifs)

Une opération entre nombres résulte toujours en une valeur de type numérique. La nature du type obtenu dépend des types des opérandes et opérateurs :
- toutes les *opérandes sont de même type* : le résultat de l’opération est du type commun.
- si *opérandes de types différents* : Les opérations en Java se font seulement si toutes les opérandes sont de même type. Ainsi, le compilateur réalise une *conversion implicite* du type des opérandes de manière à les rendre uniformément representées. Elle se fait vers le type le plus grand des types des opérandes selon la table de conversion donnée plus bas.
- Toute conversion contraire au sens donnée par cette table, doit se faire explicitement.



Exemples :

- `3 + 4 * 5` ⇒ résultat de type `int`
- Conversion `int` → `float` : dans `3 - 4.6` on convertit `3` → `3.0` ⇒ `3.0 + 4.6` ⇒ `7.6 (float)`
- `float + double` ⇒ `double`

Une conversion implicite n'est faite que lorsqu'elle n'entraîne pas de perte dans l'information stockée. Par exemple, `3` (de type `int`) peut être converti implicitement en `3.0` (de type `float`), mais `3.5` ne peut pas être convertit implicitement vers un `int` : on perdrait la partie décimale. Cependant, une conversion implicite peut faire perdre en magnitude : un `long` (64 bits) est convertit implicitement vers `float` (32 bits). Nous verrons plus tard comment la conversion implicite opère également entre types d'objets.

4.2.4 Conversions explicites (*Cast*)

La conversion entre types avec perte d'information est autorisée, lorsqu'elle est réalisée explicitement par le programmeur. Par exemple, pour convertir `3.67` vers un `int`, il suffit de d'écrire :

`(int) 3.67` ⇒ 3

ce qui donne la valeur `3` de type `int`. Une conversion explicite, appelée *cast* en Java, prend la forme `(type_cible) v`, où `type_cible` est le type vers lequel on souhaite faire la conversion. **Attention** : cette conversion n'est possible que si elle a un sens vis-à-vis des types. Ainsi, par exemple, la conversion
`(boolean) 5` est invalide car `boolean` n'est pas un type numérique.

4.2.5 Conversions simplifiées !

Nous l'avons vu, les règles de conversion implicites entre types numériques, ne sont pas forcément très intuitives, en particulier pour un débutant. Dans ce cours, nous nous limiterons à `int` et `double` parmi les types numériques. Les cas de conversion se limitent à :

- Expression avec seulement `int` ou seulement `double` ⇒ type commun en résultat.
- Expression avec `int` et `double` ⇒ `double`
- Affectation `double = int` : autorisée. Équivaut à une conversion `int` → `double`.
- Affectation `int = double` : interdite (sauf *cast* explicite. En effet, elle équivaudrait à la conversion `double` → `int`.

Exemple :

```
int x = 2; double d = 5.3;
d = d + 3.7;
x = x + 2;
d = x;
x = d;       // interdit
x = (int) d; // conversion explicite (cast) ok
```

4.2.6 Opérateurs booléans

Tous les opérateurs booléans sont binaires, sauf l'opérateur de négation `!` qui est unaire. Le résultat d'une expression booléenne est un booléan.

- `&&` (de conjonction, "et") ,
- `||` (de disjonction, "ou"),
- `!` (négation, "non")

Exemples :

- `true && false`
- `!a || (b && c)`

4.2.7 Opérateurs relationnels (de comparaison)

Permettent de comparer deux *valeurs* ou *expressions* toutes les deux de de type numérique ou `char` et renvoient une valeur booléenne en résultat.

`==` *égalité*. Renvoie `true` si les deux valeurs comparées sont égales, `false` sinon. *Exemple* : `1==4` renvoie `false`, `'a'=='a'` renvoie `true`

`<` *est plus petit*. Renvoie `true` si la première valeur est strictement plus petite que la deuxième, `false` sinon. *Exemple* : `1<4` renvoie `true`, `'a' < 'b'` renvoie `true` (ordre entre caractères ascii).

`>` *est plus grand*. Renvoie `true` si la première valeur est strictement plus grande que la deuxième, `false` sinon. *Exemple* : `1>4` renvoie `false`.

`>=` *est plus grand ou égal*. Renvoie `true` si la première valeur est plus grande ou égale que la deuxième, `false` sinon.

`<=` *est plus petit ou égal*. Renvoie `true` si la première valeur est plus petite ou égale que la deuxième, `false` sinon.

`!=` *différent*.

Exemples :

- `5 > 3`
- `a == 'b'`
- `(a >= 0) && (a <= 100)`

4.2.8 Précédence des opérateurs

Les expressions composées de plusieurs opérateurs sont évaluées de la gauche vers la droite, selon des *règles de précédence* indiquant la priorité des opérateurs les uns par rapport aux autres. Considérons par exemple :

MCours.com
Le N°1 du cours et exercices sur Internet


```
2 + 3 * 4   --> equivaut a 2 +(3*4) --> s'evalue en 14 et non pas en 20

2 + 3 > 7   --> s'evalue en false

2 + (3 > 7) --> produit une erreur
```

Dans la première, l'opérateur `*` ayant une priorité plus forte que `+`, l'expression équivaut à `2 + (3*4)` et non pas à `(2+3)*4`. Dans la deuxième, l'opérateur `>` est le moins prioritaire et ainsi l'expression est correcte. Dans la dernière, les parenthèses forcent une association entre opérateurs et opérandes qui est incorrectement typée. Voici la table des priorités, de la plus haute à la plus basse, pour les opérateurs étudiés :

<table>
<tr><td><code>!</code></td><td>négation</td></tr>
<tr><td><code>* / %</code></td><td>multiplication, division, modulo</td></tr>
<tr><td><code>+ -</code></td><td>addition, soustraction</td></tr>
<tr><td><code>> < >= <= == !=</code></td><td>comparaison</td></tr>
<tr><td><code>&& ||</code></td><td>booléens</td></tr>
<tr><td><code>=</code></td><td>affectation</td></tr>
</table>

Exercices :

1. Écrivez en Java les expressions mathématiques suivantes :
   (a) $2x - y^3$
   (b) (3 + x) + (y - z)
   (c) $1 \leq x \leq n$
   (d) $a = b = c + d$
   (e) $a \neq b$
2. Déclarez les variables nécessaires à l'évaluation des expressions plus haut, puis donnez le résultat de chacune d'entre elles.

4.3 Le typage

L'expression `3 + true` est sémantiquement incorrecte : elle est *incohérente du point de vue des types*. En effet, l'opérateur + attend deux opérands de type numérique, autrement, le calcul n'a pas de sens.

– **Le typage** est une étape d'analyse des programmes qui permet de détecter les erreurs d'incohérence entre types. Son but : reduire les erreurs introduits par le programmeur.

– **Les règles de typage** décrivent la bonne formation des instructions ou des expressions du point de vue des types. Une phrase bien formée selon les règles des typage est dite *bien typée*. Une phrase mal typée donne lieu à une *erreur de typage* pendant la compilation.

– **Le type d'une expression**. Toute expression bien typée désigne une valeur. Toute valeur appartient à un type. Donc, toute expression bien typée appartient à un type. Les règles de typage permettent d'assigner un type aux différentes formes syntaxiques d'expressions. Par exemple, une des règles de typage permet d'assigner le type `int` aux expressions utilisant des opérateurs arithmétiques et des opérandes de type `int`, long, `byte` ou `short`.

Exemples : Règle de typage de l'opérateur `*`
- e_1 `*` e_2 est une expression bien typée, et le type de son résultat est `int`,
- si e_1 et e_2 appartiennent également au type `int`.

Selon cette règle `3 * 4` est bien typée et a pour type `int` alors que `3 * true` est mal typée.

4.4 Les expressions booléennes et la Logique

Les expressions booléennes jouent un rôle important en programmation puisqu'elles interviennt de façon cruciale dans les conditionnelles et dans les boucles.

Dans une instruction de contrôle, le choix de la prochaine instruction à exécuter dépend de la valeur de l'expression boléenne dans sa condition. Il est important d'établir des conditions correctes et simples de manière à diminuer les cas des tests inutiles, les redondances et les erreurs de conception. Considérons l'instruction :

Listing 4.1 – (pas de lien)

```
if (a < b) || ((a >= b) && (c == d)) {                    // (1)
    A
} else {
    B
}
```

nous aurions pu l'écrire plus simplement par :

Listing 4.2 – (pas de lien)

```
if ((a < b) || (c == d)) {                                // (2)
    A
} else {
    B
}
```

Comment arrivons-nous à cette conclusion ? Dans `(1)`, nous remarquons que `(a < b)` et `(a >= b)` sont des conditions complémentaires : lorsque l'une est vraie, l'autre est fausse et vice-versa. Nous pouvons donc, remplacer `(a >= b)` par `!(a < b)` et `(1)` est alors équivalent à

Listing 4.3 – (pas de lien)

```
    ( a < b || ( !(a < b) && c == d ))            // (3)
```

Il y a deux cas de figure :
- Si `(a < b)`, alors, les conditions de (3), (2) et (1) sont toutes vraies.
- Si `(a < b)` faux, alors, `!(a <b)` est vrai et les conditions `(3), (2), (1)` sont vraies seulement si `(c == d)` est vraie. Sinon, les trois conditions sont fausses.

Indépendamment des valeurs de `a,b,c,d`, les trois tests sont équivalents : ils donnent vrai ou faux exactements dans les mêmes cas. Mais le test (3) et plus simple et facile à comprendre.

4.4.1 Raisonnement logique

Le raisonnement utilisé dans l'exemple précédent ne dépend pas des valeurs de `a, b, c`. En revanche, il est intéressant de voir que les expressions `(a < b)` et `(a >= b)` *sont complémentaires* (l'une est vraie lorsque l'autre est fausse), car cela nous a servi pour simplifier les tests. Remplaçons dans le test de (1),
`a < b` par `p`, `a >= b` par `!p`, et `(c==d)` par `q`. Nous obtenons :

Listing 4.4 – (pas de lien)

```
(p || ( !p && q ))
```

qui du coup semble plus facilement simplifiable. En effet, soit `p` est vrai, auquel cas, tout le test donne vrai, soit il est faux, et dans ce cas, le résultat du test dépend de la valeur de `q` : il donne vrai si `q` est vrai, faux sinon. Cela équivaut à écrire `p || q`, qui dans notre exemple, correspond bien à `(a < b || c==d)`, la simplification que nous avions trouvé. Les symboles p et q sont appelées *variables propositionnelles ou logiques*. Elles auront l'une des deux valeurs `vrai` ou `faux`.

Expressions logiques

Une expression logique (ou *booléenne*) est formée de variables logiques, des valeurs de vérité et des connecteurs logiques usuels de conjonction, disjonction, négation. Nous utiliserons les notations suivantes :

- *et* (*conjonction*) : `&&` en Java, et $\wedge$ dans nos formules logiques.
- *ou* (*disjonction*) : `||` en Java, et $\vee$ dans nos formules,
- *non* (*négation*) : `!` en Java, et $\neg$ pour nous,
- auxquels nous ajoutons le connecteur $\equiv$ d'équivalence logique.

4.4.2 Equivalences logiques

Deux expressions logiques e_1, e_2 sont équivalentes, ce que l'on note :

$$e_1 \quad \equiv \quad e_2$$

si lorsque e_1 est vraie alors e_2 est vraie aussi, et lorsque e_1 est faux, alors e_2 est faux aussi (et vice-versa). Par exemple, nous avons déjà montŕe, qu'indépendamment des valeurs de p,q, l'équivalence suivante est toujours vraie :

Listing 4.5 – (pas de lien)

```
(p || ( !p && q ))    ≡    p || q
```

Valeurs de vérité

Considérons l'expression $x + y$ où x et y sont deux entiers. La valeur de cette somme dépend de la valeur de chacun des entiers x et y. De même, toute expression logique a une valeur parmi deux possibles : vrai ou faux (on parle de *valeur de vérité*) et celle-ci dépend de la valeur des symboles qui apparaissent dans l'expression. Par exemple, la valeur de l'expression :

```
a   and   b
```

dépend des valeurs que prennent a et b. Si a et b ont chacun la valeur *vrai*, alors l'expression a and b prend la valeur *vrai*. En revanche, si au moins l'un de deux prend la valeur *faux*, alors, toute l'expression prend la valeur *faux*. Ceci est résumé dans le tableau suivant (où v signifie vrai, et f faux).

p	q	p and q
v	v	v
v	f	f
f	v	f
f	f	f

4.4.3 Tables de vérité

Pour étudier l'équivalence entre deux expressions logiques on utilise souvent les *tables de vérité*. Une table de vérité permet d'établir la valeur de vérité qui prend une expression logique, et ceci pour toutes les valeurs possibles des symboles qui en font partie. Pour construire une table de vérité on dessine une colonne pour chaque variable qui apparaît dans l'expression, et une ligne pour chaque valeur possible qu'on peut donner aux variables. La dernière colonne correspond à à l'expression qui nous intéresse et à la valeur de vérité. Nous notons `v` pour vrai, et `f`, pour faux.

Table de vérité du *et* :

p	q	p and q
v	v	v
v	f	f
f	v	f
f	f	f

Table de vérité du *ou* :

p	q	p or q
v	v	v
v	f	v
f	v	v
f	f	f

Pour établir l'équivalence entre deux expressions e_1 et e_2 on construit une table de vérité où figurent toutes les variables présentes dans les deux expressions, plus une colonne pour chacune des expressions e_1, e_2. Si pour toutes les lignes de la table les valeurs données à e_1 et e_2 sont à chaque fois les mêmes, alors, les deux expression sont équivalentes. En effet, cela veut dire que e_1, e_2 prennent la valeur vrai ou faux, exactement dans les mêmes cas.

Exemple : vérifions l'équivalence suivante :

`not(p or q)` $\equiv$ `not(p) and not(q)`

p	q	not(p or q)	not(p) and not(q)
v	v	f	f
v	f	f	f
f	v	f	f
f	f	v	v

Chaque ligne dans cette table donne la même valeur aux deux expressions. En conclusion : elles sont équivalentes.

4.4.4 Quelques équivalences utiles

Les équivalences suivantes sont très utiles dans le but de simplifier des expressions booléennes. Parfois, plutôt que de calculer la table de vérité d'une expression, il est plus intéressant de tenter de la transformer via des lois d'équivalence déjà connues.

1. Anihilation
   - p or vrai ≡ vrai
   - p and faux ≡ faux

2. Elément neutre
   - p and vrai ≡ p
   - p or faux ≡ p

3. Idempotence
   - p or p ≡ p
   - p and p ≡ p
   - not(not(p)) ≡ p

4. Lois DeMorgan
   - not(p or q) ≡ not(p) and not(q)
   - not(p and q) ≡ not(p) or not(q)

5. Simplification
   - p or (p and q) ≡ p
   - p and (p or q) ≡ p

6. Elimination de négations
   - p and (not(p) or q) ≡ p and q
   - p or (not(p) and q) ≡ p or q

7. Distribution
   - p or (q and r) ≡ (p or q) and (p or r)
   - p and (q or r) ≡ (p and q) or (p and r)

Exemple : On pourra tranformer le test `(1)` de notre premier exemple dans le test `(2)` par :

`(a < b) || ((a >= b) && (c == d)) (1)`	$\Rightarrow$	
p `||` `( !`p `&&` q`)`	$\Rightarrow$	
$p \vee (\neg(p) \wedge q)$	$\Rightarrow$	(*Distribution*)
$(p \vee \neg(p)) \wedge (p \vee q)$	$\Rightarrow$	(*Anihilation*)
$vrai \wedge (p \vee q)$	$\Rightarrow$	(*Elément neutre*)
$(p \vee q)$	$\Rightarrow$	
`(a < b) || (c == d)) (2)`		

Chapitre 5

Compléments sur l'itération

Nous conaissons déjà superficiellement les boucles `while` et `for`. Dans ce chapitre nous allons rentrer plus en détail dans la description de ces notions.

5.1 les notions de base pour l'élaboration des boucles

Toute instruction d'itération possède une condition booléenne dite *d'arrêt*, et une suite d'instructions à répéter, qu'on appelle *corps* de la boucle. Lors de chaque itération, la condition d'arrêt est testée, et selon sa valeur, on décide de répéter une nouvelle fois ou pas, les instructions du corps.
Problème 1 : Reprenons notre problème consistant à calculer la somme d'une suite de nombres entiers saisis au clavier. Le calcul s'arrête lors de la saisie du nombre zéro. Pour réaliser le calcul, nous allons répéter la saisie d'un nombre n, puis son ajout à une variable `total` qui accumule la somme de tous les nombres saisis.

Entrées : n entier à saisir
Sortie : $total$ entier
Algorithme :

1. Initialiser n par une saisie,
2. Initialiser $total$ avec 0,
3. Tant que $n \neq 0$ faire :
   (a) $total \leftarrow total + n$
   (b) Saisir n
4. Afficher la valeur de $total$.

Le pas 3 de cet algorithme est une boucle :

Tant que $n \neq 0$, faire :	*Condition d'arrêt*
(a) $total$ $\leftarrow total + n$ (b) n $\leftarrow$ nouvelle saisie	*Corps de la boucle*

Dans une boucle nous allons distinguer :
– $(n \neq 0)$ est la *condition d'arrêt*.
– n est la *variable du test* : sa valeur est testée dans la condition d'arrêt.
– $total$ est la *variable calculée* : elle détient le résultat des calculs propres au problème.

– $n, total$ sont les *variables modifiées* par le corps de la boucle.
Supposons qu'on applique l'algorithme avec saisie de 5, 3, 7, 0. L'algorithme se déroule de la manière suivante :

1.	$n \leftarrow 5$	*Initialisations*
2.	$total \leftarrow 0$	
3.	Tester $(n = 5 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 0 + 5\,; n \leftarrow 3}$
3.	Tester $(n = 3 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 0 + 5 + 3\,; n \leftarrow 7}$
3.	Tester $(n = 7 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 0 + 5 + 3 + 7\,; n \leftarrow 0}$
3.	Tester $(n = 0 \neq 0)? \Rightarrow faux \Rightarrow$	*Arrêt*
4.	Afficher $total$	15

A chaque itération, $total$ accumule la somme de sa valeur initiale (0) et des valeurs prises par $n = 5, 3, 7$.

5.1.1 Étapes d'une boucle

Toute boucle s'organise autour d'activités d'initialisation, test, calculs, etc. Nous donnons un schéma d'écriture de boucle très courant, que nous illustrons avec l'algorithme précédent. L'ordre des étapes peut varier d'une boucle à l'autre. Ce qui importe, est de ne pas omettre aucune des étapes.

1. Initialisation des variables de la boucle ($n, total$).
2. Test de la condition (sur n)
3. Dans le corps de la boucle :
   – Nouveau calcul ou traitement des données (sur $total$),
   – Modification des variables du test (n)

Dans notre exemple, l'étape d'initialisation est nécessaire pour réaliser le tout premier test sur n, mais aussi pour le premier calcul de $total$. Une mauvaise initialisation, par exemple, de $total$ à une valeur autre que 0, peut donner lieu à un calcul incorrect. L'importance de l'étape de calcul est évidente.

5.1.2 Boucles qui ne terminent pas

L'étape de modification des variables du test est primordiale pour l'arrêt de la boucle. Supposons que nous l'oublions dans notre algorithme, dont la boucle devient alors :

Tant que $n \neq 0$, faire :
– $total \leftarrow total + n$
L'application de cet algorithme ne s'arrête pas ! En effet, la valeur de n, testée par la condition d'arrêt, ne change jamais, et donc, ne peut jamais devenir égale à zéro...

1.	$n \leftarrow 5(saisie)$	*Initialisations*
2.	$total \leftarrow 0$	
3.	Tester $(n = 5 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 0 + 5}$
3.	Tester $(n = 5 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 5 + 5}$
3.	Tester $(n = 5 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 5 + 5 + 5}$
3.	Tester $(n = 5 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 5 + 5 + 5 + 5}\ldots$

5.2 Boucle `while`

5.2.1 rappels

Une boucle `while` en Java a la forme :

Listing 5.1 – (pas de lien)

```
1  while (c) {                    “tant que c est vrai, faire suiteInstructions“
2      suiteInstructions
3  }
```

où c est une expression booléenne *d'entrée* dans la boucle. Le comportement de cette boucle est :

1. c est évalué avant chaque itération.
2. Si c est vrai, on exécute $suiteInstructions$, puis le contrôle revient au point du test d'entrée (point 1).
3. Si c est faux, le contrôle du programme passe à l'instruction immédiatement après la boucle.

5.2.2 traduction de notre algorithme avec `while`

Listing 5.2 – (lien vers le code brut)

```
1  public class Somme {
2      public static void main (String[] args) {
3          int n, total;
4          // Initialisation de n,total
5          Terminal.ecrireString("Entrez un entier (fin avec 0): ");
6          n = Terminal.lireInt();
7          total = 0;
8          while ( n !=0 ) {
9              total = total + n;          // Calcul
10             Terminal.ecrireString("Entrez un entier (fin avec 0): ");
11             n = Terminal.lireInt();     // Modification variable du test
12         }
13         Terminal.ecrireStringln("La somme totale est: " + total);
14     }
15 }
```

`total` est *initialisée* à zéro, qui est l'élément neutre de l'addition ; et `n` est initialisée avec première saisie. A chaque tour de boucle, une nouvelle valeur pour `total` est calculée : c'est la somme de la dernière valeur de `total` et du dernier nombre `n` saisi (lors de l'itération précédente). De même, une nouvelle valeur pour `n` est saisie. □

Remarque : Notez que cette boucle peut ne jamais exécuter $suiteInstructions$, si avant la toute première itération, le test c est faux. Dans l'exemple 1, c'est le cas, si lors de la première saisie, n vaut 0.

Étapes d'une boucle `while`

Le programme de l'exemple 1 contient plusieurs étapes différentes : d'initialisation, test, etc, décrites dans la partie précédente. Ici, elles sont organisées de la manière suivante :

Listing 5.3 – (pas de lien)

```
1        Instructions_init             // Initialisation des variables de la boucle
2        while ( c ) {                 // Test
3            Instruction_1;...   // Calculs
4            Instruction_j;...   // Modification des variables du test
5        }
```

Il s'agit d'une structuration typique des boucles `while`, que nous reprendrons souvent. Les instructions de calcul et de modification des variables du test ne se font pas forcément dans cet ordre : selon le problème, ou selon la condition d'arrêt, on pourra les faire à des moments différents.

Voici les messages affichés par cette exécution :

```
% java Somme
Entrez un entier (fin avec 0): 5
Entrez un entier (fin avec 0): 4
Entrez un entier (fin avec 0): 7
Entrez un entier (fin avec 0): 0
La somme totale est: 16
```

5.3 Boucle `do-while`

La boucle `do-while` est une boucle `while` où les instructions du corps sont exécutées avant de tester la condition de la boucle.

Listing 5.4 – (pas de lien)

```
1    do                                        "faire suiteInstructions
2      {
3          suiteInstructions
4      }
5    while (c);                                 tant que c est vrai"
```

où c est une expression booléenne. Le comportement de cette boucle est :

1. $suiteInstructions$ est exécuté,
2. c est évaluée à la fin de chaque itération : s'il est vrai, le contrôle revient à $suiteInstructions$ (point 1).
3. Si c est faux, le contrôle du programme passe à l'instruction immédiatement après la boucle.

L'intérêt d'une boucle `do-while` est de pouvoir exécuter au moins une fois les instructions du corps avant de tester la condition d'arrêt. Cela peut éviter la duplication des instructions de modification des variables du test, qui est parfois nécessaire dans les boucles `while`. Rappelons le code de `Somme.java` :

Listing 5.5 – (lien vers le code brut)

```
1        Terminal.ecrireString("Entrez un entier (fin avec 0): ");
2        n = Terminal.lireInt(); // Saisie d'initialisation pour n
3        total = 0;
4        while ( n !=0 ) {
```

```
5             total = total + n;
6             Terminal.ecrireString("Entrez un entier (fin avec 0): ");
7             n = Terminal.lireInt(); // Nouvelle saisie de n
8         }
```

Nous sommes obligés de saisir une première valeur pour `n` avant le test d'entrée en boucle `(n != 0)`. Or, cette saisie doit être répétée lors de chaque itération. Une écriture plus élégante utilise `do-while` :

Exemple 2 : Réécriture de l'exemple 1 avec `do-while`. Dans ce code, la saisie de `n` n'est faite qu'une fois. Notez également, que l'ordre des instructions dans le corps de la boucle change : la saisie se fait avant les calculs de la même itération, et non pas pour les calculs de la prochaine.

Listing 5.6 – (lien vers le code brut)

```
1         total = 0;
2         do
3           {
4              Terminal.ecrireString("Entrez un entier (fin avec 0): ");
5              n = Terminal.lireInt();    // Saisie de n
6              total = total + n;
7           }
8         while ( n !=0 );
```

5.4 Boucle `for`

En Java, l'entête des boucles `for` incorpore les étapes d'initialisation, test, calcul et de modification des variables du test. Elle a la forme :

Listing 5.7 – (pas de lien)

```
1      for  ( initInstrs  ;  boolExpr ;  modifInstrs  ) {
2              calculInstrs
3           }
```

- `(` $initInstrs$ `;` $boolExpr$ `;` $modifInstrs$ `)` est *l'entête de la boucle* ;
- $initInstrs$ sont les *initialisations* de la boucle. Il s'agit d'une ou plusieurs instructions **séparées par des virgules**, dont le but est d'initialiser ou déclarer les variables de la boucle ;
- $boolExpr$ est l'expression booléenne *condition* de la boucle ;
- $modifInstrs$ sont les instructions de *mise à jour*. Il s'agit d'une ou plusieurs instructions séparées par des virgules, pour la mise à jour des variables de la boucle, et plus particulièrement, des variables de la condition.
- $\{calculInsts\}$ est le *corps de la boucle*. C'est un bloc d'instructions dont le but est de réaliser les calculs où traitements propres au problème.

Le comportement de cette boucle est équivalent à celui de la boucle `while` :

Listing 5.8 – (pas de lien)

```
1   {
2     initInstrs;
3     while  (boolExpr) {
4              calculInstrs;
5              modifInstrs;  }
```

```
6 }
```

1. Les instructions d'initialisation $initInstr$ sont évaluées une seule fois : avant le début des itérations.
2. $boolExpr$ est évalué avant le début de chaque itération.
3. S'il est vrai, on exécute :
   (a) les instructions $calculInstr$ du corps de la boucle,
   (b) puis celles de mise à jour des variables : $modifInstrs$,
   (c) puis, le contrôle revient au test de $boolExpr$ (point 2).
4. sinon, la boucle termine.

Exemple 3 : Calculons la somme des n premier entiers : 1 + 2 + 3 + .. + n, pour un nombre n entier positif saisit au clavier. On se donne deux variables, `i` et `somme` : à chaque tour de boucle `i` est incrémentée de 1, et `somme` accumule la somme des valeurs de `i` après un certain nombre d'itérations. On veut répéter ces actions tant que `i` est inférieur ou égal à n.

Entrées : n entier à saisir
Sortie : $somme$ entier
Algorithme :

1. Initialiser n par une saisie,
2. Initialiser $somme$ avec 0, et i avec 1, ($initInstrs$)
3. Tant que $i \leq n$ faire : ($boolExpr$)
   (a) $somme \leftarrow somme + i$ ($calculInstrs$)
   (b) $i \leftarrow i + 1$ ($modifInstrs$)
4. Afficher la valeur de $somme$.

Une traduction avec une boucle `for` est :

Listing 5.9 – (lien vers le code brut)

```
1  public class Somme_n {
2      public static void main (String[] args) {
3          int n, i, somme;
4          Terminal.ecrireString("Un entier positif? ");
5          n = Terminal.lireInt();
6          for (somme = 0, i = 1; i <= n; i = i+1) {
7              somme = somme + i;
8          }
9          Terminal.ecrireString("La somme 1+2+..+ "+ n);
10         Terminal.ecrireStringln(" = " + somme);
11     }
12 }
```

Voici une boucle `while` équivalente. Elle est une traduction presque textuelle de notre algorithme de départ :

Listing 5.10 – (lien vers le code brut)

```
1          somme = 0; i = 1;           // Initialisations
2          while (i <= n)  {
```

```
3            somme = somme + i;        // Calculs
4            i = i+1;                  // Increment
5        }
```

Une trace de l'exemple, en supposant que l'on saisit `n = 4` :

	init	*itération* 1	*itération* 2	*itération* 3	*itération* 4	*itération* 5
`(i <=n)`		`(1 <= 4)`	`(2 <= 4)`	`(3 <= 4)`	`(4 <= 4)`	`(5 <= 4)`
`somme`	0	0+1	0+1+2	0+1+2+3	0+1+2+3+4	*arrêt*
`i`	1	2	3	4	5	*arrêt*

5.4.1 Usage des boucles `for`

En Java, les boucles `for` sont plus expressives que dans la plupart de langages : avec elles, on peut écrire les mêmes boucles qu'avec un `while`. Mais, par convention, les boucles `for` sont utilisées pour faire varier une ou plusieurs variables d'*itération*, dans un intervalle de valeurs reliées entre elles, jusqu'à la vérification d'une condition. Dans l'exemple 3, la variable d'itération est `i`, qui varie de $i = 1$ jusqu'à $i \leq n$. Ces variations se font souvent par incrément ou décrément, à l'aide d'expressions `i++` ou `i--`. La boucle de l'exemple 3 devient alors :

Listing 5.11 – (pas de lien)

```
1        for (somme = 0, i = 1; i <= n; i++) {
2            somme = somme + i;
3        }
```

Exemple 4 : Calculons la puissance a^b pour deux nombres entiers a, b saisis au clavier, où $b \geq 0$. Nous faisons varier (dégressive-ment) une variable d'itération i, de $i = b$ jusqu'à $i = 1$.

Entrées : a, b entiers à saisir, $b \geq 0$.
Sortie : p entier.
Variable d'itération : $i \in [b \dots 1]$ (intervalle d'itération)
Algorithme :

1. Initialiser a, b par une saisie,
2. Initialiser p avec 1, et i avec b, $(initInstrs)$
3. Tant que $i \geq 1$ faire : $(boolExpr)$
   - $p \leftarrow p * a$ $(calculInstrs)$
   - $i \leftarrow i - 1$ $(modifInstrs)$
4. Afficher la valeur de p.

Écrite avec un `for`, cette boucle devient :

Listing 5.12 – (pas de lien)

```
1        for (p = 1, i = b; i >= 1; i--) {
2            p = p*a;
3        }
```

5.4.2 Variables locales à une boucle `for`

Dans une boucle `for`, les variables d'itération n'ont souvent d'intérêt que le temps d'exécuter la boucle. Une fois finie, c'est le résultat calculé par celle-ci qu'il est important de récupérer, afficher, etc. Dans ce cas, une bonne pratique de programmation est d'introduire ces variables de manière locale à la boucle, par une déclaration dans sa partie initialisation ($initInstrs$).

Exemple 5 : Calcul de $1 + \ldots + n$ avec la variable d'itération `i` déclarée localement :

Listing 5.13 – (pas de lien)

```
1        somme = 0;
2        for (int i = 1; i <= n; i++) {
3            somme = somme + i;
4        }
```

La portée de `i` est l'entête et le corps de la boucle. En sortie de boucle, `i` n'est plus visible. Cette zone de visibilité est bien illustrée par la traduction vers une boucle `while` (voir *comportement* d'un `for`) :

Listing 5.14 – (pas de lien)

```
1   {
2     initInstrs;
3     while  (boolExpr) {
4             calculInstrs;
5             modifInstrs; }
6   }
```

Toutes les instructions de l'entête et du corps sont entourées d'un bloc supplémentaire. C'est lui qui assure que les variables déclarées par $initInstrs$ ne seront visibles que par les instructions de la boucle.

5.4.3 Entête d'une boucle `for`

Quelques remarques sur les instructions de l'entête d'un `for` :

- Les instructions de l'entête sont toutes optionnelles (voir exemple 6.1).
- Comme dans une boucle `while`, la condition ($boolExpr$) d'une boucle `for` est évaluée avant chaque itération. Ainsi, les instructions du corps ne sont jamais exécutées si la condition est fausse dès le début.
- En revanche, les instructions d'initialisation ($initInstrs$) sont toujours exécutées, même s'il n'y a aucune itération.

Exemple 6 :

1. Pas d'instructions dans l'entête (boucle infinie). Cette boucle ne s'arrête que si `Instructions` exécute une instruction de *rupture de contrôle* (voir plus loin), ou *lève une exception* (voir chapitre sur les fonctions).

   Listing 5.15 – (pas de lien)

   ```
   1        for (;;) {
   2            Instructions
   3        }
   ```

2. `for` qui ne boucle pas. Si `n = 0`, la condition de la boucle est fausse, et il n'y a aucune itération.

Listing 5.16 – (pas de lien)

```
1        for (somme = 0, i = 1; i <= n; i++) {
2             somme = somme + i;
3        }
```

L'exécution des initialisations fait que la valeur affichée pour `somme` est correcte (0).

5.5 Instructions de rupture de contrôle : `break`

Les instructions de *rupture de contrôle* permettent la sortie d'un bloc ou d'une boucle avant d'avoir complété leur exécution. Parmi ces instructions, nous étudions `break` dans ce chapitre, et `return` dans le chapitre dédié aux fonctions. `break` est utilisée pour sortir du corps d'une boucle[1] : son utilisation en dehors des boucles (ou d'un `switch`) provoque une erreur à la compilation.

Exemple 7 : Sortie d'une boucle infinie. L'exécution de ce programme affiche les messages A, B, et F. Le message D n'est pas affiché, car il se trouve dans le corps de la boucle, après le `if` qui exécute la sortie de boucle. Si nous enlevons les commentaires pour l'affichage de C, le compilateur signale que cette instruction est inaccesible. En effet, étant juste après un `break`, elle ne sera jamais exécutée.

```
public class TestBreak {
    public static void main (String[] args) {
        int n = 1;
        while (true) {
              Terminal.ecrireStringln("A");
              if (n >= 1) {
                            Terminal.ecrireStringln("B");
                            break;
                            // Terminal.ecrireStringln("C");
      }
              Terminal.ecrireStringln("D");
        }
        Terminal.ecrireStringln("F");
}}

Java/Essais> java TestBreak
A
B
F
```

□

Exemple 8 : Le jeu suivant pose des questions à l'utilisateur. Après 2 bonnes réponses, la boucle s'arrête et l'utilisateur gagne. Si en fin de boucle, il y a moins de 2 bonnes réponses, un nouveau tour de jeu est entamé, mais il tient compte du nombre des bonnes réponses des tours précédents.

[1]Mais aussi pour sortir d'un `switch`, que nous n'étudions pas dans ces notes.

Listing 5.17 – (lien vers le code brut)

```
1  public class JeuBete {
2      public static void main (String[] args) {
3          Terminal.ecrireStringln("       ***** Jouez au jeu JAVA ****** ");
4          Terminal.ecrireStringln("Repondez par true ou false , ");
5          Terminal.ecrireStringln("et gagnez au bout de 2 bonnes reponses \n");
6          int rep =0;
7          for (boolean c;;) {
8              Terminal.ecrireString("Toute instruction finit par un ; ? ");
9              c = Terminal.lireBoolean();
10             if (!c) { rep = rep +1; }
11             if (rep == 2) {break;}
12
13             Terminal.ecrireString("L'expression (x=2 == x=3) est mal typee ?");
14             c = Terminal.lireBoolean();
15             if (!c) { rep = rep +1;}
16             if (rep == 2) {break;}
17
18             Terminal.ecrireString("Le resultat de 6/4 est 1? ");
19             c = Terminal.lireBoolean();
20             if (c) { rep = rep +1;}
21             if (rep == 2) {break;}
22
23             Terminal.ecrireString("Le resultat de 6%4 est 2 ? ");
24             c = Terminal.lireBoolean();
25             if (c) { rep = rep +1;}
26             if (rep == 2) {break;}
27             else {
28                 Terminal.ecrireString("\n **** " + rep + " bonnes reponses!");
29                 Terminal.ecrireStringln(" Encore un tour...");
30             }
31         }
32         Terminal.ecrireStringln("Bravo! Vous avez gagne!!");
33     }
34 }
```

```
Java/Essais> java JeuBete
       ***** Jouez au jeu JAVA ******
Repondez par true ou false,
et gagnez au bout de 2 bonnes reponses

Toute instruction finit par un ; ? false
L'expression (x=2 == x=3) est mal typee ? true
Le resultat de 6/4 est 1? false
Le resultat de 6%4 est 2 ? false

 **** 1 bonnes reponse!! Encore un tour...
Toute instruction finit par un ; ? true
L'expression (x=2 == x=3) est mal typee ? false
Bravo! Vous avez gagne!!
```

Remarques :
– Les instructions `break` sont à utiliser avec modération : elles peuvent rendre plus difficile la compréhension des programmes, et, la plupart du temps, on peut écrire les boucles aussi simplement sans elles.
– Lorsque `break` est exécuté par une boucle imbriquée dans d'autres boucles (voir exemple 11), il permet la sortie du corps de la boucle où il apparaît, mais pas des boucles plus externes.

5.6 Exemples

5.6.1 Boucles imbriquées

Exemple 9 : Le programme suivant affiche la table de multiplication d'un nombre entier n saisit au clavier :

Listing 5.18 – (lien vers le code brut)

```
1  public class UneTableMult {
2      public static void main (String[] args) {
3          int n;
4          Terminal.ecrireString("Un entier entre 2 et 9? ");
5          n = Terminal.lireInt();
6          Terminal.sautDeLigne();
7          Terminal.ecrireStringln("Table de " + n);
8          Terminal.ecrireStringln("**********");
9          for (int i=1; i <= 10; i++) {
10             Terminal.ecrireStringln(n + " x " + i + " = " + (n*i));
11         }
12         Terminal.sautDeLigne();
13     }
14 }
```

```
Java/Essais> java UneTableMult

Un entier entre 2 et 9? 3

Table de 3
**********
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
```

Exemple 10 : Écrire un programme qui affiche toutes les tables de multiplication de 2 à 9. On écrira deux *boucles imbriquées* :

– La boucle la plus externe fait varier entre 2 et 9 le numéro n de la table à afficher.
– Dans cette première boucle, on veut, pour chaque n différent, afficher les lignes : `n x 1`, `n x 2`, ...
– Cela revient à réaliser pour chaque pas n de la boucle externe, toute l'exécution d'une boucle plus interne, chargée d'afficher `n x i` pour i qui varie entre 1 et 10 (comme dans `UneTableMult`).
– `n` et `i` n'étant pas nécessaires en dehors des boucles, nous pouvons les déclarer localement.

Listing 5.19 – (lien vers le code brut)

```
1  public class TablesMult {
2      public static void main (String[] args) {
3          Terminal.sautDeLigne();
4          Terminal.ecrireStringln("Tables de multiplication de 2 a 9");
5          Terminal.ecrireStringln("*********************************");
6          Terminal.sautDeLigne();
7          for (int n=2; n <= 9; n++) {
8              Terminal.ecrireStringln("Table de " + n);
9              Terminal.ecrireStringln("**********");
10             for (int i=1; i <= 10; i++) {
11                 Terminal.ecrireStringln(n + " x " + i + " = " + (n*i));
12             }
13             Terminal.sautDeLigne();
14             Terminal.ecrireStringln("----------------");
15             Terminal.sautDeLigne();
16         }
17     }
18 }
```

et les premiers affichages à l'exécution :

```
Tables de multiplication de 2 a 9
*********************************

Table de 2
**********
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20

----------------

Table de 3
**********
3 x 1 = 3
```

```
3 x 2 = 6
3 x 3 = 9
........
```

Exemple 11 : Sortie d'une boucle imbriquée avec `break`. Dans cet exemple, `break` est exécuté par une boucle imbriquée, ce qui provoque l'arrêt et la sortie de la boucle qui entoure immédiatement cet instruction. Ainsi, seule la boucle sur `i` est arrêtée. Après sortie de celle-ci, le contrôle reprend dans la prochaine instruction de la boucle externe (affichage de `Terminal.ecrireStringln(" Fin boucle 1"))`. Cette boucle externe exécute plusieurs fois la boucle interne, avec à chaque fois une sortie par `break`.

Listing 5.20 – (lien vers le code brut)

```
1  public class TestBreak2 {
2      public static void main (String[] args) {
3          for (int m=1; m <= 3; m++)  {
4              Terminal.ecrireStringln("Boucle 1: m = " + m );
5              for (int i=1; i <= 10; i++)  {
6                  Terminal.ecrireStringln("  Boucle 2: i = " + i);
7                  if (true) break;
8                  Terminal.ecrireStringln("  Fin boucle 2");
9              }
10             Terminal.ecrireStringln("Fin boucle 1");
11         }
12     }
13 }
```

```
Java/Essais> java TestBreak2
Boucle 1: m = 1
  Boucle 2: i = 1
Fin boucle 1
Boucle 1: m = 2
  Boucle 2: i = 1
Fin boucle 1
Boucle 1: m = 3
  Boucle 2: i = 1
Fin boucle 1
```

5.6.2 Validation des entrées

Parfois, certains calculs ne sont définis que pour un sous-ensemble de toutes les entrées possibles au programme. Par exemple, la somme des n premiers entiers positifs $1+2+\ldots+n$, n'est défini que si $n \geq 1$. Or, un programme qui déclare n en tant que variable entière, peut saisir une valeur entière négative. Le programme de l'exemple 2, devant une entrée négative, affiche un message qui n'a pas beaucoup de sens :

Listing 5.21 – (lien vers le code brut)

```
1  public class Somme_n {
2      public static void main (String[] args) {
3          int n, i, somme;
```

```
4          Terminal.ecrireString("Un entier positif? ");
5          n = Terminal.lireInt();
6          for (somme = 0, i = 1; i <= n; i = i+1) {
7               somme = somme + i;
8          }
9          Terminal.ecrireString("La somme 1+2+..+ "+ n);
10         Terminal.ecrireStringln(" = " + somme);
11     }
12 }
```

```
Java/Essais> java Somme_n
Un entier positif? -4
La somme 1+2+..+ -4 = 0
```

Ici, la boucle `for` initialise `somme` à 0 et `i` à 1, puis teste `i <= n` avec `n` est négatif. Le test étant faux, le corps de la boucle n'est jamais exécuté et la variable `somme` reste avec sa valeur initiale 0. Modifions ce programme pour conditionner l'exécution de la boucle à la validité des données saisies, et pour signaler une erreur dans le cas contraire.

Exemple 12 : Calcul de $1 + 2 + \ldots + n$ avec message d'erreur si $n \leq 0$.

Listing 5.22 – (lien vers le code brut)

```
1  public class Somme_nValBasic {
2      public static void main (String[] args) {
3          int n, somme;
4          Terminal.ecrireString("Un entier positif? ");
5          n = Terminal.lireInt();
6          if (n > 0) {
7              somme = 0;
8              for (int i = 1; i <= n; i++) {
9                   somme = somme + i;
10             }
11             Terminal.ecrireString("La somme 1+2+..+ "+ n);
12             Terminal.ecrireStringln(" = " + somme);
13         } else {
14             Terminal.ecrireStringln("Nombre negatif: calcul impossible");
15         }
16     }
17 }
```

```
Java/Essais> java Somme_nValBasic
Un entier positif? -4
Nombre negatif: calcul impossible

/Java/Essais> java Somme_nValBasic
Un entier positif? 4
La somme 1+2+..+ 4 = 10
```

Le problème ici est qu'il faut relancer l'exécution du programme en cas d'erreur de saisie. Une meilleure solution, est de demander une nouvelle saisie via une *boucle de saisie et validation*. Elle

fera la saisie des entrées puis testera leur validité, et cela tant que celles-ci ne sont pas correctes.

Exemple 13 : Calcul de $1 + 2 + \ldots + n$ avec boucle de saisie et validation de n.

Listing 5.23 – (lien vers le code brut)

```
public class Somme_nVal {
    public static void main (String[] args) {
        int n, somme;
        do {
              Terminal.ecrireString("Un entier positif? ");
              n = Terminal.lireInt();
              if (n > 0) { break;}
              Terminal.ecrireStringln("*** Nombre negatif. Recommencez! ");
        } while (true);
        somme = 0;
        for (int i = 1; i <= n; i++) {
             somme = somme + i;
        }
        Terminal.ecrireString("La somme 1+2+..+ "+ n);
        Terminal.ecrireStringln(" = " + somme);
    }
}
```

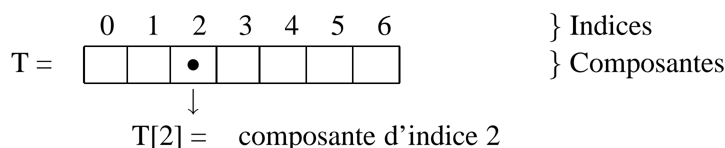
```
Java/Essais> java Somme_nVal
Un entier positif? -4
*** Nombre negatif. Recommencez!
Un entier positif? -3
*** Nombre negatif. Recommencez!
Un entier positif? 5
La somme 1+2+..+ 5 = 15
```

Chapitre 6

Tableaux

Jusqu'ici, nous avons employé les variables pour stocker les valeurs *individuelles* de types primitifs : une variable de type `int` pour stocker un entier, une variable de type `boolean` pour un booléan, etc.

Un **tableau** est *une structure regroupant plusieurs valeurs de même type*, appelées *composantes* du tableau. On peut traiter un tableau comme un tout, ou composante par composante. Traité comme un tout, on pourra le stocker dans une variable, le passer en paramètre ou le donner en résultat d'un calcul. Chaque *composante* est désignée individuellement via son *indice*, qui correspond à sa position dans le tableau, et peut être traitée comme *variable individuelle* : on pourra consulter sa valeur, la modifier, etc.



T[0], T[1], T[2], T[3], . . ., T[6] } 7 Variables (composantes)

Les tableaux sont des structures des données présentes dans tous les langages de programmation. En général, chaque langage possède un *type tableau prédéfini* avec une syntaxe spécifique.

6.1 Déclaration et création

En Java, avant d'utiliser un tableau, il faut :

1. **Déclarer** une variable de type tableau (symbole `[]`), en indiquant le type T de ses futures composantes ;

```
T [] tab;          // tab est declare tableau de T
```

2. **Créer explicitement** la structure du tableau en mémoire (opération `new`), en donnant sa taille et le type T de ses éléments. Cette taille ne pourra plus changer : en Java les tableaux sont de *taille fixe*.

```
tab = new T[taille];
```

3. L'initialisation des composantes avec des valeurs par défaut, est réalisée implicitement par l'opération de création.

Étudions plus en détail ces étapes.

Déclaration

L'instruction :

```
T [] tab;
```

déclare une variable `tab` *destinée à contenir* un tableau, dont les composantes seront de type T. Après déclaration, la variable `tab` existe, mais **n'est pas encore initialisée** à un tableau. En Java, on dira que `tab` **ne référence pas encore**[1] de tableau. Le dessin suivant montre l'état de `tab` en mémoire :

`tab` $\not\longrightarrow$

où le symbole $\not\longrightarrow$ est lu : "n'est pas initialisé".

Exemples :

```
int []  tabNum;      // tabNum est un tableau avec composantes de type int
double [] t;         // t est un tableau avec composantes de type double
String [] m;         // m est un tableau avec composantes de type String
tabNum[0] = 5;       // provoque une erreur: le tableau n'existe pas
```

Après ces déclarations, les variables `tabNum`, `t` et `m` existent, mais pas encore la suite de composantes que chacune d'entre elles pourra désigner. Par exemple, il est impossible de modifier la première composante de `tabNum` (notée `tabNum[0]`) : elle n'existe pas encore. Le compilateur signale l'erreur :

```
Test.java:7: variable tabNum might not have been initialized
    tabNum[0] = 5;
    ^
```

Création

L'opération de création :

```
new T[n];
```

réalise la création et l'initialisation d'un tableau de n composantes de type T :

1. *Allocation en mémoire* d'un espace suffisant pour stocker n composantes de type T.
2. *Initialisation* des composantes du tableau avec des valeurs par défaut.

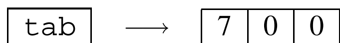
Les tableaux en Java sont de taille fixe. Une fois le tableau créé, l'espace qui lui est alloué en mémoire ne peut pas changer. Par exemple, il est impossible d'ajouter ou d'enlever des composantes d'un tableau.

Exemple 1 : Déclaration, puis création du tableau `tab` avec trois entiers.

[1] La notion de référence sera abordée dans le chapitre dédié aux objets.

```
int []  tab;          // Declaration
tab = new int[3];     // Creation
tab]0] = 7;
```

Après l'instruction de création `new`, la variable `tab` est initialisée (ou, *fait référence*) à un tableau contenant trois entiers. Après l'affectation `tab[0] = 7`, la structure du tableau en mémoire est :



Il est possible de réunir la déclaration et la création d'un tableau en une seule instruction. On pourra ainsi déclarer et créer le tableau de l'exemple 1 par :

Exemple 2 : Déclaration et création en une seule instruction.

```
int []  tab = new int[3];
```

Initialisation par une liste de valeurs

Lorsqu'un tableau est de petite taille, il est possible de l'initialiser en donnant la liste des valeurs de chaque composante. On utilise la notation $\{v_0, v_1, \ldots v_n\}$, où v_i est la valeur à donner à la composante i du tableau. Nous reprendrons souvent cette notation pour **regrouper en une seule instruction** la déclaration, la création et l'initialisation d'un tableau.

Exemple 3 : Déclaration, création et initialisation d'un tableau en une seule instruction.

```
int []  tab = {1,9,2,4};
```

Il est alors **inutile de réaliser une création explicite** via `new` : elle se fait automatiquement à la taille nécessaire pour stocker le nombre des valeurs données. En mémoire on aura :

Valeurs par défaut à la création

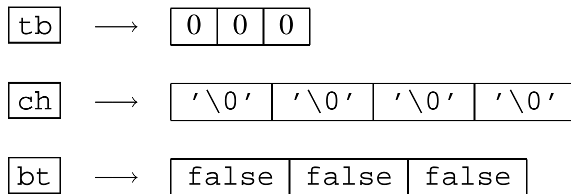
Lors de la création, les composantes d'un tableau sont initialisées avec des valeurs par défaut :
- les composantes `boolean` sont initialisées à `false`.
- les composantes numériques sont initialisées à 0.
- les composantes `char` sont initialisées au caractère nul `'\0'`.
- les composantes *référence*[2] sont initialisées à la valeur `null` (référence nulle).

Exemple 4 : Valeurs par défaut dans les composantes après création.

```
int []  tb = new int[3];
char [] ch = new char[4];
boolean [] bt = new boolean[3];
```

[2] Voir le chapitre dédié aux objets.

L'opération new initialise les composantes de ces tableaux par :

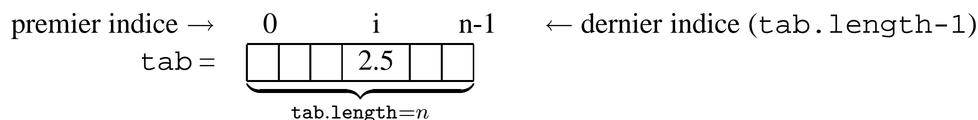


Longueur d'un tableau

La *taille* ou *longueur* d'un tableau est le nombre n des composantes qu'il contient. Supposons que tab désigne un tableau de taille n. On peut obtenir sa longueur par la notation tab.lenght. Les indices du tableau tab sont alors compris entre 0 et tab.length-1. Cette notation sera souvent employée pour fixer la valeur maximale qui peut prendre l'indice d'un tableau (voir exemple 6).

6.2 Accès aux composantes

L'accès à une composante de tableau permet de traiter cette composante comme n'importe quelle variable individuelle : on peut modifier sa valeur dans le tableau, l'utiliser pour un calcul, un affichage, etc. L'accès d'une composante se fait via son *indice ou position* dans le tableau. En Java, la première position a pour indice 0, et la dernière, a l'indice n-1 (taille du tableau moins un).



tab[i] vaut 2.5

L'accès aux composantes de tab n'a de sens que pour les indices dans l'intervalle [0,. . ., tab.length-1]. Si i est un indice compris dans cet intervalle :

- tab[i] : est un **accès à la composante de position i dans** tab. On peut consulter ou modifier cette valeur dans le tableau. *Exemple* : tab[i] = tab[i] + 1;
- **accès en dehors des bornes du tableau** tab : si j n'est pas compris entre 0 et tab.length-1, l'accès tab[j] provoque une erreur à l'exécution : l'indice j et la composante associée n'existent pas dans le tableau. Java *lève l'exception* ArrayIndexOutOfBoundsException.

Exemple 5 : Modification d'une composante, accès en dehors des bornes d'un tableau.

Listing 6.1 – (lien vers le code brut)

```
1 public class Test {
2     public static void main (String args[]) {
3         double [] tab = {1.0, 2.5, 7.2, 0.6};    // Creation
4         // Affichage avant modification
5         Terminal.ecrireString("tab[0] avant = ");
6         Terminal.ecrireDoubleln(tab[0]);
```

```
7           tab[0] = tab[0] + 4;
8           // Affichage apr\'es modification
9           Terminal.ecrireString("tab[0] apres = ");
10          Terminal.ecrireDoubleln(tab[0]);
11          //  tab[5] = 17;   // Erreur: indice en dehors des bornes
12      }
13  }
```

Ce programme affiche la valeur de la première composante de `tab`, avant et après modification.

```
Java/Essais> java Test
tab[0] avant = 1.0
tab[0] apres = 5.0
```

Si nous enlevons le commentaire de la ligne 11, l'exécution se termine par une erreur : l'indice 5 est en dehors des bornes du tableau.

```
Java/Essais> java Test
tab[0] avant = 1.0
tab[0] apres = 5.0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Test.main(Test.java:9)
```

Exemple 6 : Parcours pour affichage d'un tableau `tab`. La boucle fait varier l'indice de `tab` entre `i= 0` et `i <= tab.lenght-1`.

Listing 6.2 – (lien vers le code brut)

```
1  public class AfficheTab {
2      public static void main (String args[]) {
3          int[] tab = {10,20,30,40};
4          for (int i=0; i<= tab.length -1; i++)  {
5              Terminal.ecrireStringln("tab["+ i + "] =  " + tab[i]);
6          }
7      }
8  }
```

Ce programma affiche :

```
Java/Essais> java AfficheTab
tab[0] =  10
tab[1] =  20
tab[2] =  30
tab[3] =  40
```

Une erreur commune dans une boucle, est de fixer le dernier indice à `i <= tab.length` plutôt qu'à `i <= tab.length -1`. Si nous changeons la condition de la boucle de cette manière, l'exécution produit une erreur : l'indice `tab.length`, égal à 4 ici, n'existe pas dans `tab`.

```
Java/Essais> java AfficheTabErr
tab[0] =  10
tab[1] =  20
tab[2] =  30
tab[3] =  40
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at AfficheTabErr.main(AfficheTabErr.java:5)
```

Exemple 7 : Boucles d'initialisation et d'affichage. On souhaite initialiser un tableau avec des notes d'élèves saisies au clavier. Le programme demande le nombre de notes à saisir, et créé un tableau `lesNotes` de cette taille. Une première boucle initialise le tableau ; la boucle suivante affiche son contenu. Les itérations se font dans l'intervalle de `i=0` jusqu'à `i <= lesNotes.length-1`.

Listing 6.3 – (lien vers le code brut)

```
1  public class Notes {
2      public static void main (String args[]) {
3          int nombreNotes;
4          Terminal.ecrireString("Nombre de notes a saisir? ");
5          nombreNotes = Terminal.lireInt();
6          double [] lesNotes = new double[nombreNotes];
7          // Initialisation
8          for (int i=0; i<= lesNotes.length -1; i++)  {
9              Terminal.ecrireString("Note no. " + (i+1) + "? ");
10             lesNotes[i] = Terminal.lireDouble();
11         }
12         // Affichage
13         Terminal.sautDeLigne();
14         Terminal.ecrireStringln("Notes dans le tableau:");
15         Terminal.ecrireStringln("**********************");
16         for (int i=0; i<= lesNotes.length -1; i++)  {
17             Terminal.ecrireString("Note no. " + (i+1) + " = ");
18             Terminal.ecrireDoubleln(lesNotes[i]);
19         }
20     }
21 }
```

```
Java/Essais> java Notes
Nombre de notes a saisir? 4
Note no. 1? 7.6
Note no. 2? 11
Note no. 3? 14
Note no. 4? 5

Notes dans le tableau:
**********************
Note no. 1 = 7.6
Note no. 2 = 11.0
Note no. 3 = 14.0
Note no. 4 = 5.0
```

Affectation entre tableaux

Il est possible d'affecter une variable de type tableau à une autre autre variable, à condition qu'elles soient déclarées avec le même type de composantes. Après affectation, *les deux variables référent à un même et seul tableau en mémoire* : elles deviennent *synonymes*. Toute modification sur un composant de l'une modifie le même composant de l'autre.

Exemple 8 :

Listing 6.4 – (lien vers le code brut)

```
1          int [] t;
2          int [] m = {2,3,4,5,6};
3          t = m;    // t et m designent un meme tableau
4          Terminal.ecrireStringln("t[0] = " + t[0]);
5          Terminal.ecrireStringln("m[0] = " + m[0]);
6          // Modification de t[0]
7          t[0] = 9;
8          // Nouveaux t[0], m[0]
9          Terminal.ecrireStringln("Nouveau t[0] = " + t[0]);
10         Terminal.ecrireStringln("Nouveau m[0] = " + m[0]);
```

L'affectation de `m` dans `t` a pour *effet de bord* de rendre ces deux variables *synonymes* : elles *référent* toutes les deux au même espace mémoire, qui contient le tableau $\{2, 3, 4, 5, 6\}$ désigné par `m`. L'affectation `t[0] = 9` est ainsi "visible" lors d'un accès à `m`. Ce programme affiche :

```
t[0] = 2
m[0] = 2
Nouveau t[0] = 9
Nouveau m[0] = 9
```

Enfin, les tableaux désignés par les variables d'une affectation peuvent avoir des longueurs différentes.

Exemple 9 : Affectation entre tableaux de tailles différentes.

Listing 6.5 – (lien vers le code brut)

```
1          int [] t = {10, 20};
2          int [] m = {2,3,4,5,6};
3          Terminal.ecrireStringln("Longueur de t = " + t.length);
4          t = m;    // t contient maintenant un tableau de 5 elements
5          // Nouvelle longueur de t
6          Terminal.ecrireStringln("Nouvelle longueur de t = " + t.length);
```

Ce programme affiche :

```
Longueur de t = 2
Nouvelle longueur de t = 5
```

Égalité entre tableaux

Lorsqu'on compare deux tableaux par des tests *d'égalité* (`==`) ou *d'inégalité* (`!=`), le test porte sur *l'identité des tableaux et non sur leur contenu*. Cela signifie qu'on cherche à savoir, non pas si les tableaux contiennent les mêmes éléments, mais s'ils ont été crées par un seul et même `new`. Voyons un exemple.

Listing 6.6 – (lien vers le code brut)

```
1 public class Tab5{
2     public static void main(String[] argv){
3         int[] tab1;
4         int[] tab2;
5         int[] tab3;
6         tab1 = new int[3];
```

```
7           tab1[0] = 10;
8           tab1[1] = 20;
9           tab1[2] = 30;
10          tab2 = new int[3];
11          tab2[0] = 10;
12          tab2[1] = 20;
13          tab2[2] = 30;
14          tab3 = tab2;
15          if (tab1 == tab2){
16              Terminal.ecrireStringln("tab1 et tab2 sont egaux");
17          }else{
18              Terminal.ecrireStringln("tab1 et tab2 sont differents");
19          }
20          if (tab2 == tab3){
21              Terminal.ecrireStringln("tab2 et tab3 sont egaux");
22          }else{
23              Terminal.ecrireStringln("tab2 et tab3 sont differents");
24          }
25      }
26  }
```

L'exécution du programme donne :

```
> java Tab5
tab1 et tab2 sont différents
tab2 et tab3 sont égaux
```

Explication : `tab1` et `tab2` sont deux tableaux dont les contenus sont égaux, mais ces deux tableaux ont été créés par deux `new` différents. Il s'agit donc de deux espaces mémoires distincts. Changer l'un ne change pas l'autre. En revanche, `tab2` et `tab3` sont créés par un seul et unique `new`. Changer le contenu de l'un change aussi le contenu de l'autre. Il s'agit d'un seul et unique espace mémoire.

6.3 Exemples avec tableaux unidimensionnels

Exemple 10 : Recherche du minimum et maximum dans un tableau d'entiers. Deux variables `min` et `max` sont initialisées avec le premier élément du tableau. La boucle de recherche des minimum et maximum, compare chaque élément avec ces deux valeurs. La comparaison se fait à partir du deuxième élément : c'est pourquoi l'indice i débute à `i=1`.

Listing 6.7 – (lien vers le code brut)

```
1   public class minMax {
2       public static void main (String args[]) {
3           int n;
4           Terminal.ecrireString("Combien des nombres a` saisir? ");
5           n = Terminal.lireInt();
6           int [] tab = new int[n];
7           // Initialisation
8           for (int i=0; i<= tab.length -1; i++)  {
9                Terminal.ecrireString("Un entier? ");
10               tab[i] = Terminal.lireInt();
```

```
11          }
12          // Recherche de min et max
13          int min = tab[0];
14          int  max = tab[0];
15          for (int i=1; i<= tab.length -1; i++)  {
16              if (tab[i] < min) { min = tab[i];}
17              if (tab[i] > max) { max = tab[i];}
18          }
19          Terminal.ecrireStringln("Le minimum est: " + min);
20          Terminal.ecrireStringln("Le maximum est: " + max);
21      }
22  }
```

Voici une exécution du programme :

```
Java/Essais> java minMax
Combien des nombres a‘ saisir? 5
Un entier? 5
Un entier? 2
Un entier? -6
Un entier? 45
Un entier? 3
Le minimum est: -6
Le maximum est: 45
```

Exemple 11 : Gestion de notes. Nous modifions le programme de l'exemple 7 afin de calculer la moyenne des notes, la note minimale et maximale, et le nombre de notes supérieures ou égales à 10. Nous reprenons la boucle de recherche du minimum et maximum de l'exemple 10.

Listing 6.8 – (lien vers le code brut)

```
1   public class Notes {
2       public static void main (String args[]) {
3           int nombreNotes;
4           Terminal.ecrireString("Nombre de notes a saisir? ");
5           nombreNotes = Terminal.lireInt();
6           double [] lesNotes = new double[nombreNotes];
7           // Initialisation
8           for (int i=0; i<= lesNotes.length -1; i++)  {
9                Terminal.ecrireString("Note no. " + (i+1) + "? ");
10               lesNotes[i] = Terminal.lireDouble();
11          }
12          double min = lesNotes[0];
13          double max = lesNotes[0];
14          double somme = 0;
15          int sup10 = 0;
16          for (int i=0; i<= lesNotes.length -1; i++)  {
17              if (lesNotes[i] < min) { min = lesNotes[i];}
18              if (lesNotes[i] > max) { max = lesNotes[i];}
19              if (lesNotes[i] >= 10) { sup10++;}
20              somme = somme + lesNotes[i];
21          }
22          Terminal.ecrireString("La moyenne des notes est: ");
```

```
23          Terminal.ecrireDoubleln(somme/nombreNotes);
24          Terminal.ecrireStringln("Le nombre de notes >= 10 est: " + sup10);
25          Terminal.ecrireStringln("La note minimum est: " + min);
26          Terminal.ecrireStringln("La note  maximum est: " + max);
27      }}
```

```
Java/Essais> java Notes
Nombre de notes a' saisir? 4
Note no. 1? 5
Note no. 2? 8
Note no. 3? 10
Note no. 4? 15
La moyenne des notes est: 9.5
Le nombre de notes >= 10 est: 2
La note minimum est: 5.0
La note  maximum est: 15.0
```

Exemple 12 : Inversion d'un tableau de caractères. La boucle d'inversion utilise deux variables d'itération i et j, initialisées avec le premier et le dernier élément du tableau. A chaque tour de boucle, les éléments dans i et j sont échangés, puis i est incrémenté et j décrémenté. Il y a deux cas d'arrêt possibles selon la taille du tableau : s'il est de taille impair, alors l'arrêt se produit lorsque i=j ; s'il est de taille pair, alors l'arrêt se fait lorsque `j < i`. En conclusion, la boucle doit terminer si `i >= j`.

Listing 6.9 – (lien vers le code brut)

```
1   public class Inversion {
2       public static void main (String[] args) {
3           int n;
4           char [] t;
5           Terminal.ecrireString("Combien de caracteres a saisir? ");
6           n = Terminal.lireInt();
7           t = new char[n];
8           // Initialisation
9           for (int i=0; i<= t.length -1; i++) {
10                Terminal.ecrireString("Un caractere:  ");
11                t[i] = Terminal.lireChar();
12          }
13          // Affichage avant inversion
14          Terminal.ecrireString("Le tableau saisi: ");
15          for (int i=0; i<= t.length -1; i++) {
16                Terminal.ecrireChar(t[i]);
17          }
18          Terminal.sautDeLigne();
19          // Inversion: arret si (i >= j)
20          int i,j;
21          char tampon;
22          for (i=0, j= t.length-1 ; i < j; i++, j--) {
23                 tampon = t[i];
24                 t[i] = t[j];
25                 t[j] = tampon;
26          }
```

```
27          // Affichage final
28          Terminal.ecrireString("Le tableau inverse: ");
29          for (int k=0; k<= t.length -1; k++) {
30                Terminal.ecrireChar(t[k]);
31          }
32          Terminal.sautDeLigne();
33      }
34  }
```

Un exemple d'exécution :

```
Java/Essais> java Inversion
Combien de caracteres a saisir? 5
Un caractere:  s
Un caractere:  a
Un caractere:  l
Un caractere:  u
Un caractere:  t
Le tableau saisit: salut
Le tableau inverse: tulas
```

6.4 Le tri par sélection

Les programmes de tri de tableaux sont couramment utilisés dans la gestion des données. Nous présentons un algorithme simple de tri, le *tri par sélection*. Il est appelé ainsi car, lors de chaque itération, il sélectionne l'élément le plus petit parmi ceux *restant à trier* et le met à sa place dans le tableau, en *l'échangeant avec l'élément* qui s'y trouve. Nous illustrons cette méthode sur le tableau $\{4, 5, 1, 9, 8\}$.

La première fois, l'algorithme sélectionne le plus petit du tableau, qui est 1, et l'échange avec l'élément qui se trouve à la première place (4) :

$$\begin{array}{rl} & \{\underline{4}, 5, \boxed{1}, 9, 8\} \\ \text{après échange} \Rightarrow & \{\boxed{1}, 5, \underline{4}, 9, 8\} \end{array}$$

Au bout de cette première itération, le premier élément est trié : c'est le plus petit de tout le tableau. Les éléments restant à trier sont ceux à partir de la 2ème position. La fois suivante, le plus petit parmi eux (4), est sélectionné et échangé avec l'élément se trouvant à la deuxième place (5) :

$$\begin{array}{rl} & \{1, \underline{5}, \boxed{4}, 9, 8\} \\ \text{après échange} \Rightarrow & \{1, \boxed{4}, \underline{5}, 9, 8\} \end{array}$$

Après la deuxième itération, les deux premiers éléments sont triés : la première place contient le plus petit, la deuxième, contient le deuxième plus petit. L'algorithme finit lorsqu'il ne reste plus qu'un seul seul élément à trier, qui se trouve nécessairement à sa place : la dernière. Terminons l'application de l'algorithme :

$$\begin{array}{rcl} & \{1, 4, \boxed{5}, 9, 8\} : & \text{5 est échangé avec lui-même} \\ \Rightarrow & \{1, 4, 5, \underline{9}, \boxed{8}\} : & \text{8 est échangé avec 9} \\ \Rightarrow & \{1, 4, 5, 8, 9\} : & \text{Le tableau est trié} \end{array}$$

Algorithme de tri par sélection :

Entrée : un tableau tab taille n
Sortie : le tableau tab trié.

Pour chaque indice i de `tab` compris dans [0, . . ., n-2], faire :

1. Sélectionner le plus petit élément parmi ceux d'indices [i, . . ., n-1], et déterminer son indice `Im`.
2. Échanger les éléments `tab[Im]` et `tab[i]`.

□

Le pas de sélection du plus petit élément se fait également avec une boucle. La technique employée est similaire à celle de l'exemple 10, mais ici, la recherche ne se fait pas sur tout le tableau, mais seulement sur la partie restant à trier lors de chaque itération.

Exemple 13 : Tri par sélection.

Listing 6.10 – (lien vers le code brut)

```
1  public class triSelection {
2      public static void main (String args[]) {
3          int n;
4          int [] tab;
5          Terminal.ecrireString("Nombre de entiers a trier? ");
6          n = Terminal.lireInt();
7          tab = new int[n];
8          // Initialisation de tab
9          for (int i = 0; i <= tab.length -1; i++)  {
10              Terminal.ecrireString("Un entier? ");
11              tab[i] = Terminal.lireInt();
12         }
13         // Tri
14         for (int i = 0; i <= tab.length -2; i++) {
15             // Recherche du min dans [i .. tab.lentgh -1]
16             int Im = i;
17             for (int j = i+1 ; j <= tab.length -1; j++) {
18                 if (tab[j] < tab[Im]) {Im = j;}
19             }
20             // Echange de tab[i] avec le min trouve
21             int tampon = tab[i];
22             tab[i] = tab[Im];
23             tab[Im] = tampon;
24         }
25         // Affichages
26         Terminal.ecrireString("Tableau trie: ");
27         for (int i = 0; i <= tab.length -1; i++)  {
28              Terminal.ecrireString("  ");
29              Terminal.ecrireInt(tab[i]);
30         }
31         Terminal.sautDeLigne();
32     }
33 }
```

Voici un exemple d'exécution :

```
Java/Essais> java triSelection
Nombre d'entiers a trier? 6
Un entier? 10
Un entier? 2
Un entier? 7
Un entier? 21
Un entier? 8
Un entier? 5
Tableau trie:  2  5  7  8  10  21
```

Nous présentons une trace partielle de ce programme, sur le tableau `tab = {4,5,1,9,8 }`. Elle montre, pour chaque pas de boucle, la position de l'indice i, et celle de l'indice I_m du plus petit élément parmi ceux restant à trier :

Itération 1 : `i = 0`, `Im` = 2.

$i_\downarrow$		$I_{m\downarrow}$		
4	5	**1**	9	8

Après échange :

1 ‖	5	4	9	8

Itération 2 : `i = 1`, `Im` = 2.

	$i_\downarrow$	$I_{m\downarrow}$		
1	**5**	**4**	9	8

Après échange :

1	4 ‖	5	9	8

Itération 3 : `i = 2`, `Im` = 2.

		I_m $i_\downarrow$		
1	4	**5**	9	8

Après échange : rien ne change.

Itération 4 : `i = 3`, `Im` = 4.

			$i_\downarrow$	$I_{m\downarrow}$
1	4	5	**9**	**8**

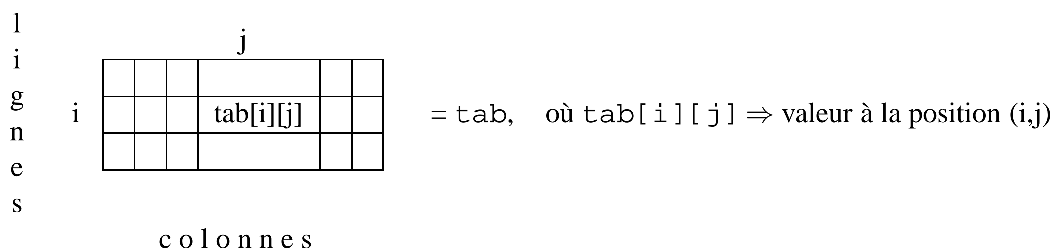
Après échange, il ne reste plus qu'un élément : le tableau est donc trié.

1	4	5	8 ‖	9

6.5 Tableaux à deux dimensions

Les tableaux vus jusqu'ici sont des *tableaux à une dimension* : conceptuellement tous les éléments se trouvent dans une seule ligne (ou colonne, cela revient au même). Les tableaux à plusieurs dimensions sont utiles dans la modélisation des données, mais ce sont les tableaux à deux dimensions qui sont de loin les plus utilisés en informatique. Nous concentrons notre étude à leur cas.

Un *tableau à deux dimensions*, ou *matrice*, représente un rectangle composé de lignes et de colonnes. Chaque élément stockée dans le tableau est adressé par sa position, donnée par sa ligne et sa colonne. En Java, si `tab` est un tableau à deux dimensions, l'élément de ligne i et colonne j est désigné par `tab[i][j]`.



Déclaration

En Java, un tableau de n dimensions et composantes de type T est déclaré par :

```
T [] [] $\ldots$ []  tab;     // n fois le symbole []
```

Chaque ajout du symbole [] permet d'obtenir une dimension supplémentaire :

```
int [] t;           // une dimension
int [] [] m;        // deux dimensions
char [] [] [] p;    // trois dimensions
```

Création

Comme avant, nous utilisons l'opération `new` de création de tableaux, en donnant en plus du type des éléments, la taille de chaque dimension.

Exemple 14 : Création d'un tableau d'entiers à deux dimensions, avec 3 lignes et 5 colonnes. Après création, nous modifions l'élément de la ligne 1 et colonne 2 par `t[1][2] = 7`.

```
int [][] t = new int [3][5];     //  3 lignes, 5 colonnes
t[1][2] = 7;
```

Nous pouvons imaginer `t` comme un rectangle avec 3 lignes et 5 colonnes. Par convention, la première dimension est celle des lignes, et la deuxième, celle des colonnes. Comme avant, les indices débutent à 0. Lors de sa création, les éléments du tableau sont initialisés à 0 par défaut.

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	7	0	0
2	0	0	0	0	0

Longueur d'une dimension

Il est également possible d'obtenir la longueur d'une des dimensions du tableau : nombre des lignes pour la première dimension, nombre de colonnes pour la deuxième. La notation :

- `t.length` : donne la longueur de la première dimension, c.a.d., le nombre de lignes du tableau.
- `t[i].length` : donne la longueur de la ligne i de t, autrement dit, le nombre de colonnes de cette ligne.

Exemple 15 :

```
int [][] t = new int [3][5];           //  3 lignes, 5 colonnes
Terminal.ecrireIntln(t.length);        // affiche 3
Terminal.ecrireIntln(t[1].length);     // affiche 5
```

Initialisation

En Java, les tableaux à plusieurs dimensions sont définis à l'aide de tableaux des tableaux. Ce sont des tableaux dont les composantes sont elles-mêmes des tableaux. Comme avant, il est possible d'initialiser une matrice en donnant la liste de ses composantes. Par exemple, un tableau de 3 lignes et 4 colonnes sera initialisé par une suite de trois tableaux, un pour chaque ligne. Chacun des trois tableaux sera composé de 4 éléments : c'est le nombre de colonnes dans chacune des lignes.

Exemple 15 : Initialisation d'une matrice (de 3 lignes et 4 colonnes), par énumération de ses composantes.

```
int [][] tab = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
int [] t = tab[1];
for (int j = 0; j<= t.length-1; j++) {
    Terminal.ecrireInt(t[j]);
}
Terminal.sautDeLigne();
```

Cet exemple nous permet de comprendre qu'un tableau à deux dimensions de tailles n et m, est en réalité un tableau unidimensionnel de n lignes, où chaque ligne est un tableau de m composantes. Le programme affiche :

```
Java/Essais> java Test
5678
```

Parcours des matrices

Les parcours des matrices se font souvent avec des boucles imbriquées : une boucle externe pour parcourir les lignes, pour des indices compris entre 0 et mat.lenght-1, et une boucle interne qui, pour

chaque ligne, fait le parcours des éléments dans toutes les colonnes de cette ligne. Les indices des colonnes seront alors compris entre 0 et mat[i].lenght-1.

Exemple 16 : Initialisation d'une matrice par saisie de sa taille et de ses éléments.

Listing 6.11 – (lien vers le code brut)

```
1  public class initMatrice {
2      public static void main (String args[]) {
3          int n,m;
4          Terminal.ecrireString("Nombre de lignes? ");
5          n = Terminal.lireInt();
6          Terminal.ecrireString("Nombre de colonnes? ");
7          m = Terminal.lireInt();
8          int [][] mat = new int [n][m];
9          // Initialisation
10         for (int i=0; i<= mat.length -1; i++) {
11             for (int j=0; j<= mat[i].length -1; j++) {
12                 Terminal.ecrireString("Element (" + i + ", " + j + ")? ");
13                 mat[i][j] = Terminal.lireInt();
14             }
15         }
16     }
17 }
```

Ce programme affiche :

```
Java/Essais> java initMatrice
Nombre de lignes? 2
Nombre de colonnes? 3
Element (0, 0)? 1
Element (0, 1)? 2
Element (0, 2)? 3
Element (1, 0)? 4
Element (1, 1)? 5
Element (1, 2)? 6
```

Exemple 17 : Gestion des notes par élèves. Ce programme initialise une matrice de n élèves avec m notes par élève, puis calcule dans un tableau de taille n, la moyenne de chaque élève. Les notes de l'élève i se trouvent à la ligne i de la matrice `notes`, alors que sa moyenne est dans `moyennes[i]`.

Listing 6.12 – (lien vers le code brut)

```
1  public class matriceNotes {
2      public static void main (String args[]) {
3          int n,m;
4          Terminal.ecrireString("Nombre d'eleves? ");
5          n = Terminal.lireInt();
6          Terminal.ecrireString("Nombre de notes par eleve? ");
7          m = Terminal.lireInt();
8          double [][] notes = new double [n][m];
9          double [] moyennes = new double [n];
```

```
10          // Initialisation
11          for (int i=0; i<= notes.length -1; i++) {
12              Terminal.ecrireStringln("Notes pour l'eleve "+ (i+1)+ "?");
13              for (int j=0; j<= notes[i].length -1; j++) {
14                   Terminal.ecrireString("   Note "+ (j+1) + "? ");
15                   notes[i][j] = Terminal.lireDouble();
16               }
17          }
18         // Calcul des moyennes
19          for (int i=0; i<= notes.length -1; i++) {
20              for (int j=0; j<= notes[i].length -1; j++) {
21                   moyennes[i] = moyennes[i] + notes[i][j];
22               }
23              moyennes[i] = moyennes[i]/notes[i].length;
24          }
25          //Affichages
26          for (int i=0; i<= moyennes.length -1; i++) {
27              Terminal.ecrireString("Moyenne de l'eleve "+ (i+1) + "= ");
28              Terminal.ecrireDoubleln(moyennes[i]);
29          }
30      }
31  }
```

Ce programme affiche :

```
Java/Essais> java matriceNotes
Nombre d'eleves? 3
Nombre de notes par eleve? 2
Notes pour l'eleve 1?
   Note 1? 2
   Note 2? 2
Notes pour l'eleve 2?
   Note 1? 6
   Note 2? 17
Notes pour l'eleve 3?
   Note 1? 10
   Note 2? 15
Moyenne de l'eleve 1= 2.0
Moyenne de l'eleve 2= 11.5
Moyenne de l'eleve 3= 12.5
```

Chapitre 7

Sous-programmes

7.1 Fonction : notion mathématique

Voici la définition mathématique classique du mot fonction :
Définition Soient A et B deux ensembles. Une fonction f définie sur A (ou de domaine A, ou d'espace de départ A, ou de domaine de définition A) à valeurs dans B (ou de codomaine B, ou d'espace d'arrivée B, ou d'espace image B) est une correspondance qui, à tout élément x de A, fait correspondre un élément et un seul, noté f(x), de B. Cet élément f(x) est appelé résultat de l'application de f à l'élément x (parfois image de x par f).

Remarque : Il ne faut pas confondre la fonction f, qui est un élément de l'ensemble des fonctions de A dans B (en général noté A -> B), et le résultat de l'application de f à un argument x, qui est un élément de B. Dans certains cours de mathématiques, lorsque l'on ne s'intéresse pas aux fonctions en tant que telles mais seulement aux résultats de leurs applications, on parle parfois de la fonction f(x).

La notion de fonction pose plusieurs questions. La première est celle de la *calculabilité*, c'est à dire la possibilité de calculer la valeur f(x) pour une valeur x de A donnée. On peut définir certaines fonctions sans pour autant avoir de moyen de réaliser le calcul correspondant.

Par exemple, la fonction qui à un numéro de département associe le nombre de personnes présentes actuellement dans ce département. Cette fonction a un sens parfaitement compréhensible, et ce nombre de personnes existe. Simplement, il n'y a aucun moyen réaliste pour calculer ce nombre. Même l'INSEE, l'Institut National des Statistiques ne peut faire qu'un calcul approximatif.

L'informaticien s'intéresse presque exclusivement à des fonctions calculables, et parmi celles-ci, plus spécialement à celles qu'un ordinateur peut calculer dans des conditions acceptables, c'est à dire en un temps limité et avec des moyens en rapport avec les enjeux du calcul. Par exemple, si l'on fait un programme de prévision météorologique pour les cinq jours qui viennent, si on a une fonction qui calcule très précisément ces prévisions en fonction des relevés de stations météo, mais qu'il faut deux mois pour calculer cette fonction, elle n'a aucun intérêt pratique.

Une autre questions qui se pose est celle du langage que l'on emploie pour définir une fonction. Nous aborderons cette question d'abord du point de vue des mathématiques qui fournissent une base théorique pour les constructions offertes par les langages informatiques pour définir des fonctions.

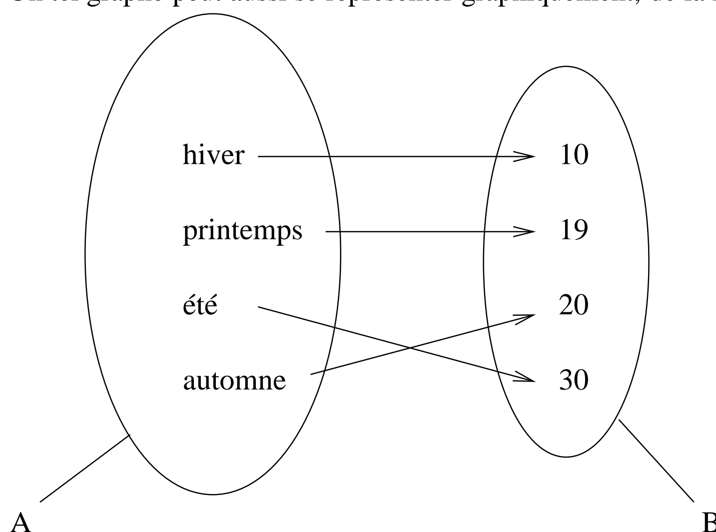
En mathématiques, il existe plusieurs méthodes pour définir une fonction. Voici celles qui sont le plus couramment employées.

7.1.1 Construire le graphe

Le graphe d'une fonction f de A dans B est l'ensemble des couples `(x,y)` où `x` est un élément de A, `y` est un élément de B et `y=f(x)`. Si le domaine A est fini, on peut indiquer explicitement quel est l'élément de B qui correspond à un élément donné de A. Dans ce cas, on peut définir la fonction par son graphe de manière effective.

Soit l'ensemble A = {hiver, printemps, été, automne}. On peut définir une fonction `max_temp` par le graphe suivant : `{(hiver,10),(printemps,19),(été,30),(automne,20)}`.

Un tel graphe peut aussi se représenter graphiquement, de la façon suivante :



graphe de f

Calcul du résultat de l'application

Calculer le résultat de l'application d'une fonction définie par un graphe à un argument donné `v` consiste simplement à chercher le couple `(v,y)` dans le graphe et en extraire la valeur `y` qui est la valeur de `f(v)`. Par définition de la notion de fonction, il existe un seul couple commençant par `v` dans le graphe de la fonction `f`.

7.1.2 Donner une expression

La fonction peut être définie par une expression dont le calcul donne la valeur de la fonction en tout point de son domaine de définition. Pour cela, il faut choisir un nom, disons x, pour désigner un élément quelconque du domaine. Ce nom est appelé variable en mathématiques.

Par exemple on peut définir la fonction `f(x)=3*x+2` Dans cette fonction, `x` est la variable. C'est un nom que l'on donne pour désigner un élément quelconque de l'ensemble A. L'expression est `3*x2+`. Cette expression contient la variable, des constantes (2 et 3) et des opérations (+ et *). Ces opérations sont elles-memes des fonctions, définies avant `f`.

Le nom de la variable n'a pas d'importance. Par exemple, les deux définitions suivantes définissent une seule et même fonction : `f(x)=x+3` et `f(y)=y+3`.

Calcul du résultat de l'application

Pour calculer la valeur d'une fonction `f` définie par une expression pour une valeur donnée de l'ensemble de définition A, il faut remplacer dans l'expression la variable par cette valeur. Cela donne

une expression dans laquelle il n'y a plus d'inconnue. On peut réaliser le calcul pour obtenir une valeur de l'ensemble B.

Par exemple, pour calculer la valeur de `f` définie par `f(x)=3*x+2` pour la valeur 5, on remplace `x` par 5 dans l'expression `3*x+2`. Cela donne l'expression `3*5+2` dans laquelle il n'y a pas d'inconnue et qu'on peut calculer pour obtenir le résultat `17`. On en conclut que +f(5)=17+.

Par rapport à la définition au moyen d'un graphe, la définition de fonction par une expression a l'avantage de permettre de définir des fonctions dont le graphe est infini. Par exemple, la fonction `f(x)=3*x+2` est définie pour tout l'ensemble des entiers relatifs, qui est un ensemble infini. Il y a donc une infinité de couples dans le graphe de cette fonction.

L'ordre des calculs n'est pas important, c'est à dire que le résultat obtenu à la fin est le même quel que soit l'ordre de réalisation du calcul. Il existe d'ailleurs un certains nombres de lois définissant l'équivalence d'expressions, permettant de réaliser les calculs plus simplement (factorisation, associativité, commutativité).

7.1.3 Utiliser une construction par cas

Une fonction peut aussi ne pas être définie de la même façon suivant les valeurs de la variable. On utilise différentes expressions pour différents sous-ensemble de l'ensemble de définition A. On parle de fonction définie par morceau.

Voici quelques fonctions définies à l'aide de constructions par cas, écrites de différentes manières.

1. abs(x) = si x $\leq$ 0 alors x sinon (- x)
2. `par_morceaux`(x) = x + 1 si x $\leq$ 1
x + 4 si x > 1 et x $\leq$ 100
x + 2 si x > 100
3. continue (x) = sin(x) / x si x $\neq$ 0
1 sinon

Pour qu'une telle définition soit valide, il faut que les différents cas soient mutuellement exclusifs, c'est à dire que pour une valeur donnée, il n'y ait qu'une définition.

Calcul du résultat de l'application

Pour calculer le résultat de l'application d'une fonction définie par morceau pour une valeur `v` donnée, il faut d'abord déterminer quel cas s'applique à cette valeur puis effectuer le calcul de l'expression donnée pour ce cas.

7.1.4 Utiliser la récursion

Les moyens déjà vus ne sont pas suffisants pour décrire par exemple la fonction factorielle de N dans N. La suite des valeurs de factorielle est souvent décrite comme suit :

```
0! = 1
n! = 1 * 2 * 3 * ... * (n-1) * n
```

L'écriture ci-dessus, même si elle est évocatrice, n'est en rien effective. Il est impossible d'écrire un algorithme contenant des points de suspension ! Pour définir une fonction calculant effectivement la factorielle d'un entier, il faut autoriser l'emploi du nom de la fonction en cours de définition dans l'expression qui la définit. Une telle définition sera dite récursive. Nous avons déjà rencontré des définitions récursives dans les descriptions de syntaxes. Voici une définition effective de la fonction factorielle : fact (n) = si n=0 ou n=1 alors 1 sinon n * fact (n-1)

Suivons le calcul de fact(3) en utilisant le symbole >> pour abréger la phrase “se simplifie en”. fact(3) >> `3 * fact(2)` >> `3 * 2 * fact(1)` >> `3 * 2 * 1` >> 6 Le calcul a pu être mené à bien parce que le calcul de fact(1) est fait directement, sans appel de la fonction fact.

Pour être correcte, toute définition récursive de fonction, disons f, doit utiliser au moins une construction par cas avec au moins un cas dans lequel l'expression ne comporte pas la fonction f, de façon à permettre l'arrêt des calculs. Ce cas est appelé cas de base de la récursion.

Un autre exemple bien classique de fonction récursive est la fonction de Fibonnacci, définie par :

fib (n) = 1 si n =0 ou n=1
fib(n -1) + fib(n-2) sinon

Un autre exemple tout aussi célèbre est celui de la fonction d'Ackerman définie sur N$\times$ N par :

Ack(m, p) = p+1 si m=0
Ack (m-1,1) si p = 0
Ack(m-1, Ack(m,p-1)) sinon

Le lecteur est invité à calculer les valeurs Ack(m,p), pour les premières valeurs de m et p.

7.2 Fonction dans un programme

Il arrive que l'on fasse des calculs dans un programme. On peut vouloir exprimer ces calculs sous forme de fonction pour deux raisons :

1. éviter les répétitions. Si un même calcul apparaît à de multiples reprises dans un programme, en définissant une fonction, on n'écrit ce calcul qu'une fois, lorsque l'on décrit la fonction. Ensuite, chaque calcul consiste à utiliser cette fonction.
2. rendre le programme plus clair, plus lisible, en donnant un nom au calcul.

En java, il existe quelques fonctions prédéfinies appelées opérateurs. Ce sont les fonctions les plus courantes utilisées par chaque type de donnée. Pour les types numériques, ce sont les quatre opérations arithmétiques usuelles, pour le type boolean, les connecteurs logiques, et chaque type primitif possède ainsi quelques fonctions.

Il est possible d'écrire des fonctions dans un programme en utilisant la construction Java qui s'appelle *méthode*. Voyons un exemple simple : la fonction qui calcule la valeur absolue d'un nombre.

Si l'on cherche à caractériser mathématiquement cette fonction, on dira que c'est une fonction dont le domaine de définition est l'ensemble des entiers relatifs et les valeurs appartiennent à l'ensemble des entiers naturels (une valeur absolue est toujours positive ou nulle). Puis, on donnera une définition par cas :
$abs(x) = x$ si $x \geq 0$
$abs(x) = -x$ si $x < 0$

Voyons maintenant comment on peut décrire la fonction en Java. Nous avons déjà vu au chapitre 5 un programme qui calcule la valeur absolue, sans utiliser de fonction. Nous le rappelons ici, avant de voir le programme avec fonction.

Listing 7.1 – (lien vers le code brut)

```
1 public class ValAbs {
2     public static void main (String args[]) {
3         int x, abs;
4         Terminal.ecrireString("Donnez un entier: ");
5         x = Terminal.lireInt();
```

```
6          if (x > 0) {
7              abs = x;
8          } else if (x < 0) {
9              abs = -x;
10         } else {
11             abs = 0;
12         }
13         Terminal.ecrireStringln("La valeur absolue est " + abs);
14     }
15 }
```

Avec une fonction, le programme peut s'écrire comme suit.

Listing 7.2 – (lien vers le code brut)

```
1  public class ValAbsFunc {
2      static int valeurAbsolue(int n){
3          int res;
4          if (n > 0) {
5              res = n;
6          } else if (n < 0) {
7              res = -n;
8          } else {
9              res = 0;
10         }
11         return res;
12     }
13     public static void main (String args[]) {
14         int  x, abs;
15         Terminal.ecrireString("Donnez un entier: ");
16         x = Terminal.lireInt();
17         Terminal.ecrireStringln("La valeur absolue est " + valeurAbsolue(x));
18     }
19 }
```

Dans ce programme, il y a d'abord la définition d'une méthode appelée `valeurAbsolue`, puis utilisation de cette méthode dans le corps du programme (méthode `main`).

Voyons de plus près la définition. Elle comprend une ligne d'entête suivie d'un bloc (rappel : un bloc est une séquence d'instructions entre accolades). La ligne d'entête, `static int valeurAbsolue(int x)`, comprend plusieurs informations :

- le mot-clé `static`, nécessaire, qu'on n'expliquera pas dans ce chapitre. Son rôle sera détaillé plus tard dans le cours.
- le type `int` : c'est le type du **résultat** du calcul.
- le nom `valeurAbsolue` : c'est le nom de la méthode, qui est au libre choix du programmeur, comme un nom de variable.
- entre parenthèses, deux informations : le type (`int`) et le nom `n` du paramètre de la fonction. Il s'agit de la valeur dont on cherche la valeur absolue.

Le bloc qui suit la ligne d'entête est un bloc d'instructions tout à fait classique, sauf qu'il utilise une nouvelle instruction appelée `return`. Cette instruction a pour effet de terminer l'exécution de la méthode. Elle est suivie du résultat renvoyé par la méthode, c'est à dire le résultat de la fonction. Dans notre exemple, on a calculé la valeur absolue de `n` et mis le résultat dans une variable locale `res`. Cette variable est locale au bloc.

A l'intérieur du bloc de définition d'une fonction, on a le droit d'utiliser le nom du paramètre défini dans l'entête (`n` apparaît plusieurs fois dans la méthode).

En ce qui concerne l'utilisation de la méthode, la terminologie propose deux variantes : on peut parler **d'appel** de la méthode ou **d'envoi de message**. On voit l'utilisation de la méthode `valeurAbsolue` dans la ligne :

```
Terminal.ecrireStringln("La valeur absolue est " + valeurAbsolue(x));
```

L'appel de méthode proprement dit est : `valeurAbsolue(x)`, c'est à dire une expression composée du nom de la fonction suivie de la valeur de son paramètre entre parenthèses. Il s'agit de calculer la valeur absolue du contenu de la variable `x`.

7.3 Notion de paramètre

Dans une fonction, on veut exprimer un calcul qui dépend d'une ou plusieurs valeurs susceptibles de varier dans un domaine. Ces valeurs, en mathématiques sont appelées les variables de la fonction. On évitera d'utiliser cette terminologie puisqu'en programmation, on utilise le mot variable pour autre chose. On parle de *paramètres* de la fonction.

Une fonction est un calcul dans lequel il y a des inconnues. Tant que ces inconnues restent inconnues, on ne peut pas effectuer le calcul et connaître son résultat.

Une fonction est une moulinette qui prend une ou plusieurs choses en entrée et ressort une purée en sortie. Une purée, pas deux. Par exemple, supposons qu'on mette des carottes et des patates dans la moulinette. Il en ressort une purée carotte-patate et non deux purées, une de carotte et une de patates.

Les inconnues, les carottes et les patates, ce sont les paramètres de la fonction. On donne un nom à ces paramètres.

Par exemple, quand on décrit : `f(x)=3*x+2`, `x` est un nom qu'on donne à une inconnue. Tant qu'on ne sait pas ce que vaut `x`, le calcul reste impossible. Le nom qu'on donne est arbitraire. Si on écrit : `f(y)=3*y+2`, c'est toujours la même fonction.

En Java, en plus du nom, il faut donner un type aux paramètres, ce qui permet de vérifier que le calcul est possible. Par exemple, pour la fonction `f`, `x` peut être du type `int`.

L'application, l'exécution, l'appel de la fonction pourra se faire en donnant une valeur à `x` et cette valeur devra être du bon type.

Il faut également donner le type du résultat, le type de la purée produite. En touillant de l'int avec les touillettes multiplication et addition, on obtient un int. Le type doit être cohérent avec la valeur calculée.

Dans le corps de la fonction, c'est à dire dans le bloc qui suit l'entête, on peut utiliser le nom des paramètres pour dénoter une valeur inconnue au moment où l'on écrit la fonction. Cette valeur sera connue au moment de l'utilisation de la fonction, au moment de l'appel.

Nous sommes habitués à manipuler des fonctions mathématiques, c'est à dire des fonctions numériques ayant des paramètres et résultats entiers ou réels. En informatique, on peut utiliser des fonctions pour tous les types possibles.

Par exemple, on peut écrire des fonctions intéressantes utilisant des caractères. Nous illustrons cela avec une fonction qui calcule si un caractère est une lettre en majuscule ou non. Pour comprendre comment il fonctionne, il faut savoir que Java utilise un codage appelé *Unicode* où toutes les lettres majuscules (sauf celles qui ont des accents, mais généralement, on n'utilise pas les majuscules accentuées) sont contiguës. Cela signifie que, dans l'ordre du type char, entre deux majuscules, il n'y a que des majuscules.

Listing 7.3 – (lien vers le code brut)

```
1  public class TestMajuscule{
2      static boolean estMajuscule(char c){
3          return (c >= 'A') && (c <='Z');
4      }
5      public static void main (String args[]) {
6          char x;
7          Terminal.ecrireString("Donnez un caractere: ");
8          x = Terminal.lireChar();
9          if (estMajuscule(x)){
10             Terminal.ecrireStringln("C'est une majuscule");
11         }else{
12             Terminal.ecrireStringln("Ce n'est pas une majuscule");
13         }
14     }
15 }
```

Peut-être certains d'entre vous auraient préféré écrire la fonction de la façon suivante :

Listing 7.4 – (pas de lien)

```
1      static boolean estMajuscule(char c){
2          boolean res;
3          if ((c >= 'A') && (c <='Z')){
4              res = true;
5          }else{
6              res = false;
7          }
8          return res;
9      }
```

C'est strictement équivalent, simplement moins efficace et moins élégant. En effet, quand la condition `(C>='A') && (C<='Z')` vaut `true`, la fonction `estMajuscule` renvoie la valeur `true`, et quand la condition vaut `false`, la fonction renvoie la valeur `false`. Donc, elle renvoie toujours la valeur de la condition, le `if` ne sert à rien.

7.4 Résultat d'une fonction

Une fonction sert à calculer un résultat qui est une valeur du type qui apparaît le premier dans l'entête. Nous avons vu deux exemples, celui de la valeur absolue qui renvoie un résultat de type int, le test de majuscule qui renvoie un résultat booléen (vrai ou faux, le caractère est une majuscule).

Dans ces deux exemples, nous avons calculé cette valeur dans une variable locale appelée `res`. Puis, sur la dernière ligne de la méthode, nous avons renvoyé cette valeur au moyen de l'instruction `return`.

Il est possible de renvoyer la valeur calculée dès qu'on la connaît, sans passer par le stockage dans une variable locale.

Listing 7.5 – (lien vers le code brut)

```
1      static int valeurAbsolue(int n){
2          if (n > 0) {
3              return n;
```

```
4         } else if (n < 0) {
5             return -n;
6         } else {
7             return 0;
8         }
9     }
```

La seule contrainte qui existe est que dans tous les cas, la fonction doit se terminer par un return. Par exemple, si l'on écrit :

Listing 7.6 – (lien vers le code brut)

```
1     static int valeurAbsolue(int n){
2         int res;
3         if (n > 0) {
4             res = n;
5         } else if (n < 0) {
6             return -n;
7         } else {
8             return 0;
9         }
10    }
```

Dans le cas où `n` est supérieur à 0, il n'y a pas d'instruction `return`. C'est une erreur, le compilateur s'en aperçoit et donne un message :

```
> javac ValAbsFunc3.java
ValAbsFunc3.java:11: missing return statement
    }
    ^
1 error
```

7.5 Fonction à plusieurs paramètres

Une fonction peut comporter plusieurs inconnues. Le calcul n'est possible que si on donne une valeur à toutes les inconnues.

Par exemple : `f(x,y)=2*x+3*x*y+y+1`. Dans les fonctions prédéfinies de Java, il y en a qui ont plusieurs paramètres :

- les fonctions arithmétiques `+`, `*`, `-` et `/` sont des fonctions qui prennent deux paramètres de type `int` (ou un autre type numérique) et renvoie un résultat de type `int`.
- les fonctions de comparaison `<`, `>`, `<=` (notation Java pour $\leq$), `>=` (notation Java pour $\geq$), `!=` (notation Java pour $\neq$), et `=` sont des fonctions qui prennent deux paramètres de même type et renvoient un résultat de type `boolean`.
- `&&` (*et* logique), `||` (*ou* logique) sont des fonctions qui prennent deux paramètres booléens et rendent un résultat booléen.

Ces fonctions prédéfinies ont la particularité de pouvoir être utilisées avec une notation spéciale, dite notation infixe, dans laquelle la fonction apparaît entre ses deux paramètres. Lorsqu'on écrit de nouvelles fonctions non prédéfinies, la notation est un peu différente, les paramètres sont donnés entre parenthèses à la définition de la fonction comme à l'appel.

MCours.com
Le N°1 du cours et exercices sur Internet

Listing 7.7 – (lien vers le code brut)

```
1        static int f(int x, int y){
2            return 2*x+3*x*y+y+1;
3        }
```

et l'appel à cette fonction (c'est à dire l'utilisation de la fonction en donnant des valeurs aux inconnues `x` et `y`) s'écrit comme suit : `f(45,12)`.

Prenons un autre exemple. Nous venons de mentionner qu'il existe des fonctions prédéfinies pour les connecteurs logique `et` et `ou`. Il n'y en a pas pour le connecteur `implique` (le connecteur le plus pénible de la logique des propositions). Nous allons écrire la fonction `implique`.

Rappelons la table de vérité de ce connecteur :

p	q	p => q
true	true	true
true	false	false
false	true	true
false	false	true

Une première façon d'écrire la fonction consiste à coder directement la table de vérité avec des `if` imbriqués.

Listing 7.8 – (lien vers le code brut)

```
1  public class Implique1{
2      static boolean implique(boolean a, boolean b){
3          if (a){
4              if (b){
5                  return true;     // a et b vrais
6              }else{
7                  return false;    // a vrai, b faux
8              }
9          }else{
10             return true;     // a vrai, b vrai ou faux
11         }
12     }
13     public static void main (String args[]) {
14         Terminal.ecrireString("true => false vaut: ");
15         Terminal.ecrireBooleanln(implique(true,false));
16         Terminal.ecrireString("false => true vaut: ");
17         Terminal.ecrireBooleanln(implique(false,true));
18     }
19 }
```

Une autre façon de coder la fonction serait d'utiliser une formule qui nous assure que `a => b` est toujours équivalent à `(non a) ou b`. On peut utiliser le non (`!`) et le ou (`||`) qui existent en Java.

Listing 7.9 – (lien vers le code brut)

```
1  public class Implique2{
2      static boolean implique(boolean a, boolean b){
3          return (!a) || b;
4      }
5      public static void main (String args[]) {
6          Terminal.ecrireString("true => false vaut: ");
7          Terminal.ecrireBooleanln(implique(true,false));
```

```
8           Terminal.ecrireString("false => true vaut: ");
9           Terminal.ecrireBooleanln(implique(false, true));
10      }
11  }
```

Comme le montre le programme, pour utiliser la fonction `implique`, on donne son nom suivi de deux expressions de type boolean entre parenthèses, séparées par des virgules : `implique(true,false)`. Le résultat est une valeur de type bolean, c'est à dire `true` ou `false`, si bien que l'appel de fonction peut être utilisé comme condition d'un `if`.

D'autres appels possibles :
- `implique(5<3*2, 'C' = 'd')`
- `implique(5<3*2, implique(True,13/=12))`
- `implique(x,y)` si `x` et `y` sont des variables de type boolean.
- `implique(x,5<y*2)` si `x` est une variable de type boolean et `y` une variable de type `int`.

7.6 Appel de fonction

Il est très important de comprendre que les paramètres donnés à une fonction à l'appel sont des valeurs. Les paramètres sont un moyen de communiquer entre le programme et le sous-programme. La fonction `f` est une moulinette. L'appel de fonction `f(carotte,patate)` désigne ce qui sort de la moulinette (à savoir la purée). On peut utiliser `f(carotte,patate)` à tous les endroits du programme où on a besoin de purée. Il faut bien distinguer la fonction (une moulinette) de ce que son usage produit (une purée).

Lors de l'exécution de l'appel de fonction, le passage de valeurs est automatique : les valeurs données entre parenthèses sont stockées dans des cases mémoire allouées aux paramètres.

Détaillons un exemple d'appel de fonction pas à pas.

Listing 7.10 – (lien vers le code brut)

```
1   public class Divisible{
2       static boolean estDivisiblePar(int a, int b){
3           if (a % b == 0){
4               return true;
5           }else{
6               return false;
7           }
8       }
9       public static void main(String[] args){
10          int x = 35;
11          int y = 2;
12          if (estDivisiblePar(x, 3 * y - 1)){
13              Terminal.ecrireStringln("divisible");
14          }
15      }
16  }
```

Il y a dans ce programme un seul appel de fonction, à savoir `estDivisiblePar(x, 3 * y - 1))` et c'est cet appel dont nous allons détailler l'exécution pas à pas.

1. la première étape est le calcul de la valeur des paramètres.

- calcul de la valeur de `x` (valeur du premier paramètre). Au moment de l'appel de fonction, la variable `x` contient la valeur 35.
- calcul de la valeur de `3 * y - 1` (valeur du deuxième paramètre). Au moment de l'appel, la variable `y` contient la valeur 2. Cette valeur est utilisée à la place de `y` dans l'expression qui devient `3 * 2 - 1`. Cette expression vaut `5`.

2. de la mémoire est allouée pour les deux paramètres `a` et `b` et les valeurs 35 et 5 sont stockées respectivement dans ces deux cases mémoires.
3. le corps de la fonction, c'est à dire le bloc des lignes 4 à 9, est exécuté dans l'ordre.
   - ligne 4 : la condition `a % b == 0` est évaluée. On va chercher dans la mémoire la valeur de `a` (35) et celle de `b` (5) et on calcule `35 % 5`, c'est à dire le reste de la division entière de 35 par 5. Ce reste vaut 0 qui est égal à 0. Donc la condition vaut la valeur `true`.
   - la condition étant vraie, c'est le premier bloc entre `if` et `else` qui est exécuté, c'est à dire la ligne 5. Il y a là une instruction `return` suivi du résultat calculé par la fonction. Ce résultat est `true`. L'effet de cette instruction est de terminer le calcul de la fonction.
4. la mémoire donnée au paramètres `a` et `b` est libérée. Les deux paramètres cessent d'exister (jusqu'au prochain appel).
5. la valeur de l'appel `estDivisiblePar(x, 3 * y - 1))` est `true`. Cette valeur est utilisée dans l'instruction `if` de la ligne 13 : la condition est vraie, le message de la ligne 14 est affiché.

Deux erreurs à ne pas faire :

- penser qu'on ne peut donner comme paramètres de la fonctions que deux variables de nom `a` et `b`. Si on déclare deux variables `a` et `b` elles n'ont rien à voir avec les deux paramètres `a` et `b`. En particulier, des espaces mémoires différents seront attribués, même si le nom est identique.
- penser qu'il faut donner une valeur à `a` et `b` par un `Terminal.lireInt()` ou une affectation. Ce n'est ni nécessaire ni possible. L'attribution d'une valeur à `a` et `b` se fait automatiquement lors de l'appel, comme montré dans l'exemple.

7.7 Fonction partielle

Certaines fonctions ne sont pas définies sur tout le type de leurs paramètres.

Par exemple, la division entière est une fonction qui prend deux entiers et rend un entier, mais elle n'est pas définie pour tous les couples d'entiers que l'on peut lui donner. Elle n'est pas définie pour un diviseur nul. 17/0 n'est pas défini.

On dit que la division est une fonction partielle.

Que se passe-t-il en Java quand on utilise la division hors de son domaine de définition ? Vous pouvez le voir en compilant et en exécutant le programme suivant :

Listing 7.11 – (lien vers le code brut)

```
1 public class DivZero{
2     public static void main(String[] args){
3         int x = 0;
4         Terminal.ecrireIntln(17/x);
5     }
6 }
```

A la compilation, il n'y a pas de problème : le programme est bien typé ; il est considéré comme correct (si on avait écrit directement `17/0`, le compilateur aurait produit une erreur, mais en utilisant la variable `x`, on le trompe facilement).

A l'exécution, il se produit une erreur, avec un message :

```
> java DivZero
java.lang.ArithmeticException: / by zero
        at DivZero.main(DivZero.java:5)
```

En termes Java, on dit qu'une exception a été levée. Le programme s'arrête.

Lorsqu'on écrit ses propres fonctions, on veut parfois obtenir le même comportement. On veut que le programme s'arrête lorsqu'un appel à la fonction est fait avec des valeurs pour les paramètres qui sont hors de son domaine de définition.

Prenons un exemple simple : la fonction factorielle. Cette fonction est définie seulement sur les nombres positifs. Pas sur tous les entiers. Nous allons définir une nouvelle erreur et la lever lorsque le paramètre donné est négatif.

Listing 7.12 – (lien vers le code brut)

```
1  public class Factorielle{
2      static int factorielle(int n){
3          int res = 1;
4          if (n<0){
5              throw new PasDefini();
6          }
7          for (int i = 1; i<=n; i++){
8              res = res * i;
9          }
10         return res;
11     }
12     public static void main(String[] argv){
13         int x;
14         Terminal.ecrireString("Entrez un nombre (petit): ");
15         x = Terminal.lireInt();
16         Terminal.ecrireIntln(factorielle(x));
17     }
18 }
19 class PasDefini extends Error{
20 }
```

Essayez ce programme et voyez ce qui se produit quand on entre au clavier un nombre négatif.

Définir la nouvelle erreur consiste à écrire une nouvelle classe avec la clause `extends Error`. Nous verrons beaucoup plus tard ce que cela signifie exactement. Pour l'instant, il suffit d'utiliser cela comme une incantation dans laquelle vous changerez à volonté le nom de la classe en remplaçant `PasDefini` par autre chose.

Dans le programme, le déclenchement de l'erreur se fait au moyen de l'instruction `throw new PasDefini()`. Là encore, dans un premier temps, nous utiliserons cela comme une incantation, avant de voir ce que cela signifie.

On verra plus tard dans l'année qu'il est possible de lever une exception à la place d'une erreur, ce qui permet au programme de corriger l'erreur plutôt que de s'arrêter purement et simplement. Pour l'instant, nous allons seulement créer des erreurs, qui, contrairement aux exceptions, ne peuvent pas être récupérées.

7.8 Méthodes faisant des entrées-sorties

Nous avons présenté longuement comment on peut utiliser une méthode pour coder une fonction en Java. Il existe des méthodes qui ne correspondent pas à des fonctions parce qu'elles ne calculent pas un résultat. Ce sont des sous-programmes qui font des effets, notamment des sorties à l'écran.

Les méthodes de `Terminal` telles que `ecrireString` et `ecrireInt` sont de bons exemples de telles méthodes. Elles produisent une affichage mais ne renvoient pas de valeur. On n'utilise jamais ces méthodes à gauche d'une affectation ou comme condition d'un `if` ou comme portion d'un calcul.

Ces méthodes correspondent à ce qu'on appelle *procédure* dans les langages impératifs ou procéduraux.

Voyons un exemple de méthode qui n'est pas une fonction.

Listing 7.13 – (lien vers le code brut)

```
1  public class AfficheTable{
2      static void afficheTable(int[] t){
3          Terminal.ecrireChar('+');
4          for (int i=0; i<t.length; i++){
5              Terminal.ecrireString("---+");
6          }
7          Terminal.sautDeLigne();
8          Terminal.ecrireChar('|');
9          for (int i=0; i<t.length; i++){
10             Terminal.ecrireString(" " + t[i] + " |");
11         }
12         Terminal.sautDeLigne();
13         Terminal.ecrireChar('+');
14         for (int i=0; i<t.length; i++){
15             Terminal.ecrireString("---+");
16         }
17         Terminal.sautDeLigne();
18     }
19     public static void main(String[] args){
20         int[] ex = {1,5,8,9,7};
21         afficheTable(ex);
22     }
23 }
```

La méthode `afficheTable` affiche de façon pseudo-graphique les tableaux contenant des entiers à un chiffre. Dans la définition de la méthode, il y a deux différences avec les fonctions :

– dans l'entête, au lieu de déclarer le type de la valeur calculée, il y a le mot-clé `void` qui signifie précisément que la méthode ne calcule pas de valeur. En anglais, *void* signifie *vide*.
– dans le corps de la méthode, il n'y a pas nécessairement d'instruction `return`.

On voit dans l'exemple que l'appel à la méthode (`afficheTable(ex);`) est utilisé seul sur une ligne, alors que cela n'aurait pas de sens d'appeler une fonction seule sur une ligne.

Il existe enfin des méthodes hybrides qui font à la fois des entrées-sorties et qui calculent un résultat. C'est le cas par exemple de `Terminal.lireInt()`. Cette méthode renvoie une valeur de type `int`. On l'emploie souvent à droite d'une affectation (par exemple `x = Terminal.lireInt()`). Mais ce n'est pas à proprement parler une fonction parce que le résultat ne dépend pas que de la valeur des paramètres. Elle dépend aussi des entrées au clavier.

Chapitre 8

Les exceptions

8.1 Introduction : qu'est-ce qu'une exception ?

De nombreux langages de programmation de haut niveau possèdent un mécanisme permettant de gérer les erreurs qui peuvent intervenir lors de l'exécution d'un programme. Le mécanisme de gestion d'erreur le plus répandu est celui des exceptions. Nous avons déjà abordé le concept d'exception dans le cours sur les fonctions : lorsqu'une fonction n'est pas définie pour certaines valeur de ses arguments on lève une exception en utilisant le mot clef `throw`. Par exemple, la fonction factorielle n'est pas définie pour les nombres négatifs, et pour ces cas, on lève une exception :

Listing 8.1 – (lien vers le code brut)

```
1  class Factorielle {
2      static int factorielle(int n){
3          int res = 1;
4          if (n<0){
5              throw new PasDefini();
6          }
7          for(int i = 1; i <= n; i++) {
8              res = res * i;
9          }
10         return res;
11     }
12 }
13
14 class PasDefini extends Error {}
```

Une exception signale une erreur comme lorsqu'un nombre négatif est passé en argument à la fonction factorielle. Jusqu'ici, lever une exception signifiait interrompre définitivement le programme avec l'affichage d'un message d'erreur décrivant l'exception à l'écran. Cependant, il est de nombreuses situations où le programmeur aimerait gérer les erreurs sans que le programme ne s'arrête définitivement. Il est alors important de pouvoir intervenir dans le cas où une exception a été levée. Les langages qui utilisent les exceptions possèdent toujours une construction syntaxique permettant de “rattraper” (ou “récupérer”) une exception, et d'exécuter un morceau de code spécifique à ce traitement d'erreur.

Examinons maintenant comment définir, lever et récupérer une exception en Java.

8.2 Définir des exceptions

Afin de définir une nouvelle sorte d'exception, on crée une nouvelle classe en utilisant une déclaration de la forme suivante :

```
class NouvelleException extends ExceptionDejaDefinie {}
```

On peut remarquer ici la présence du mot clé **extends**, dont nous verrons la signification dans un chapitre ultérieur qui traitera de l'héritage entre classes. Dans cette construction, `NouvelleException` est le nom de la classe d'exception que l'on désire définir en "étendant" `ExceptionDejaDefinie` qui est une classe d'exception déjà définie. Sachant que `Error` est prédéfinie en Java, la déclaration suivante définit la nouvelle classe d'exception `PasDefini` :

```
class PasDefini extends Error {}
```

Il existe de nombreuses classes d'exceptions prédéfinies en Java, que l'on peut classer en trois catégories :

- Celles définies par extension de la classe `Error` : elles représentent des erreurs critiques qui ne sont pas censées être gérées en temps normal. Par exemple, une exception de type `OutOfMemoryError` est levée lorsqu'il n'y a plus de mémoire disponible dans le système. Comme elles correspondent à des erreurs critiques elles ne sont pas normalement censées être récupérées et nous verrons plus tard que cela permet certaines simplifications dans l'érciture de méthodes pouvant lever cette exception.
- Celles définies par extension de la classe `Exception` : elles représentent les erreurs qui doivent normalement être gérées par le programme. Par exemple, une exception de type `IOException` est levée en cas d'erreur lors d'une entrée sortie.
- Celles définies par extension de la classe `RuntimeException` : elles représentent des erreurs pouvant éventuellement être gérées par le programme. L'exemple typique de ce genre d'exception est `NullPointerException`, qui est levée si l'on tente d'accéder au contenu d'un tableau qui a été déclaré mais pas encore créé par un `new`.

Chaque nouvelle exception entre ainsi dans l'une de ces trois catégories. Si on suit rigoureusement ce classement (et que l'on fait fi des simplifications évoquées plus haut), l'exception `PasDefini` aurait due être déclarée par :

```
class PasDefini extends Exception {}
```

ou bien par :

```
class PasDefini extends RuntimeException {}
```

car elle ne constitue pas une erreur critique.

8.3 Lever une exception

Lorsque l'on veut lever une exception, on utilise le mot clé `throw` suivi de l'exception à lever, qu'il faut avoir créée auparavant avec la construction **new** `NomException()` Ainsi lancer une exception de la classe `PasDefini` s'écrit :

```
throw  new PasDefini();
```

Lorsqu'une exception est levée, l'exécution normale du programme s'arrête et on saute toutes les instructions jusqu'à ce que l'exception soit rattrapée ou jusqu'à ce que l'on sorte du programme. Par exemple, si on considère le code suivant :

Listing 8.2 – (lien vers le code brut)

```
1  public class Arret {
2      public static void main(String [] args) {
3          int x = Terminal.lireInt();
4          Terminal.ecrireStringln("Coucou 1");        // 1
5          if (x >0){
6              throw new Stop();
7          }
8          Terminal.ecrireStringln("Coucou 2");        // 2
9          Terminal.ecrireStringln("Coucou 3");        // 3
10         Terminal.ecrireStringln("Coucou 4");        // 4
11     }
12 }
13 class Stop extends RuntimeException {}
```

l'exécution de la commande `java Arret` puis la saisie de la valeur 5 (pour la variable `x`) produira l'affichage suivant :

```
Coucou 1
Exception in thread "main" Stop
        at Arret.main(Arret.java:7)
```

C'est-à-dire que les instructions 2, 3 et 4 n'ont pas été exécutées. Le programme se termine en indiquant que l'exception `Stop` lancée dans la méthode `main` à la ligne 7 du fichier `Arret.java` n'a pas été rattrapée.

8.4 Rattraper une exception

8.4.1 La construction `try catch`

Le rattrapage d'une exception en Java se fait en utilisant la construction :

Listing 8.3 – (pas de lien)

```
1    try {
2        ... // 1
3    } catch (UneException e) {
4        ... // 2
5    }
6    .. // 3
```

Le code 1 est normalement exécuté. Si une exception est levée lors de cette exécution, les instructions restantes dans le code 1 sont abandonnées. Si la classe de l'exception levée dans le bloc 1 est `UneException` alors le code 2 est exécuté (car l'exception est récupérée). Dans le code 2, on peut faire référence à l'exception en utilisant le nom donné à celle-ci lorsque l'on nomme sa classe. Ici le nom est `e`. Dans le cas où la classe de l'exception n'est pas `UneException`, le code 2 **et le code 3** sont sautés. Ainsi, le programme suivant :

Listing 8.4 – (lien vers le code brut)

```
1  public class Arret2 {
2      public static void P () {
3          int x = Terminal.lireInt();
4
5          if (x >0){
6              throw new Stop();
7          }
8      }
9      public static void main(String [] args) {
10         Terminal.ecrireStringln("Coucou 1");          // 1
11
12         try {
13             P ();
14             Terminal.ecrireStringln("Coucou 2");      // 2
15         } catch (Stop e){
16             Terminal.ecrireStringln("Coucou 3");      // 3
17         }
18         Terminal.ecrireStringln("Coucou 4");          // 4
19     }
20 }
21 class Stop extends RuntimeException {}
```

produit l'affichage suivant lorsqu'il est exécuté et que l'on saisit une valeur positive :

```
Coucou 1
Coucou 3
Coucou 4
```

On remarquera que l'instruction 2 n'est pas exécuté (du fait de la levée de l'exception dans `P`.

Si on exécute ce même programme mais en saisissant une valeur négative on obtient :

```
Coucou 1
Coucou 2
Coucou 4
```

car l'exception n'est pas levée.

En revanche le programme suivant, dans lequel on lève une exception `Stop2`, qui n'est pas récupérée.

Listing 8.5 – (lien vers le code brut)

```
1  public class Arret3 {
2      public static void P () {
3          int x = Terminal.lireInt();
4
5          if (x >0){
6              throw new Stop2();
7          }
8      }
9      public static void main(String [] args) {
10         Terminal.ecrireStringln("Coucou 1");          // 1
11         try {
```

```
12              P ();
13              Terminal.ecrireStringln("Coucou 2");     // 2
14          } catch (Stop e){
15              Terminal.ecrireStringln("Coucou 3");     // 3
16          }
17          Terminal.ecrireStringln("Coucou 4");         // 4
18      }
19  }
20  class Stop extends RuntimeException {}
21  class Stop2 extends RuntimeException {}
```

produit l'affichage suivant lorsqu'il est exécuté et que l'on saisit une valeur positive :

```
Coucou 1
Exception in thread "main" Stop2
        at Arret3.P(Arret3.java:7)
        at Arret3.main(Arret3.java:15)
```

8.4.2 Rattraper plusieurs exceptions

Il est possible de rattraper plusieurs types d'exceptions en enchaînant les constructions `catch` :

Listing 8.6 – (lien vers le code brut)

```
1   public class Arret3 {
2       public static void P () {
3           int x = Terminal.lireInt();
4
5           if (x >0){
6               throw new Stop2();
7           }
8       }
9       public static void main(String [] args) {
10          Terminal.ecrireStringln("Coucou 1");         // 1
11          try {
12              P ();
13              Terminal.ecrireStringln("Coucou 2");     // 2
14          } catch (Stop e){
15              Terminal.ecrireStringln("Coucou 3");     // 3
16          }
17          } catch (Stop2 e){
18              Terminal.ecrireStringln("Coucou 3 bis");// 3bis
19          }
20          Terminal.ecrireStringln("Coucou 4");         // 4
21      }
22  }
23  class Stop extends RuntimeException {}
24  class Stop2 extends RuntimeException {}
```

A l'exécution, on obtient (en saisissant une valeur positive) :

```
Coucou 1
Coucou 3 bis
Coucou 4
```

8.5 Exceptions et méthodes

8.5.1 Exception non rattrapée dans le corps d'une méthode

Comme on l'a vu dans les exemples précédents, lorsqu'une exception est levée lors de l'exécution d'une méthode et qu'elle n'est pas rattrapée dans cette méthode, elle "continue son trajet" à partir de l'appel de la méthode. Même si la méthode est sensée renvoyer une valeur, elle ne le fait pas :

Listing 8.7 – (lien vers le code brut)

```
1  public class Arret {
2      static int lance(int x) {
3          if (x < 0) {
4              throw new Stop();
5          }
6          return x;
7      }
8      public static void main(String [] args) {
9          int y = 0;
10         try {
11             Terminal.ecrireStringln("Coucou 1");
12             y = lance(-2);
13             Terminal.ecrireStringln("Coucou 2");
14         } catch (Stop e) {
15             Terminal.ecrireStringln("Coucou 3");
16         }
17         Terminal.ecrireStringln("y vaut "+y);
18     }
19 }
20 class Stop extends RuntimeException {}
```

A l'exécution on obtient :

```
Coucou 1
Coucou 3
y vaut 0
```

8.5.2 Déclaration `throws`

Lorsqu'une méthode lève une exception définie par extension de la classe `Exception` il est nécessaire de préciser au niveau de la déclaration de la méthode qu'elle peut potentiellement lever une exception de cette classe. Cette déclaration prend la forme **`throws`** `Exception1, Exception2, ...` et se place entre les arguments de la méthode et l'accolade ouvrant marquant le début du corps de la méthode. On notera que cette déclaration n'est pas obligatoire pour les exceptions de la catégorie `Error` ni pour celles de la catégorie `RuntimeException`.

8.6 Exemple résumé

On reprend l'exemple de la fonction factorielle :

Listing 8.8 – (lien vers le code brut)

```
1  class Factorielle {
2      static int factorielle(int n) throws PasDefini { // (1)
3          int res = 1;
4          if (n<0){
5              throw new PasDefini();                     // (2)
6          }
7          for(int i = 1; i <= n; i++) {
8              res = res * i;
9          }
10         return res;
11     }
12     public static void main (String [] args) {
13         int x;
14         Terminal.ecrireString("Entrez un nombre (petit):");
15         x = Terminal.lireInt();
16         try {                                          // (3)
17             Terminal.ecrireIntln(factorielle(x));
18         } catch (PasDefini e) {                        // (3 bis)
19             Terminal.ecrireStringln("La factorielle de "
20                                 +x+" n'est pas definie !");
21         }
22     }
23 }
24 class PasDefini extends Exception {}                   // (4)
```

Dans ce programme, on définit une nouvelle classe d'exception `PasDefini` au point (4). Cette exception est levée par l'instruction `throw` au point (2) lorsque l'argument de la méthode est négatif. Dans ce cas l'exception n'est pas rattrapée dans le corps et comme elle n'est ni dans la catégorie `Error` ni dans la catégorie `RuntimeException`, on la déclare comme pouvant être levée par factorielle, en utilisant la déclaration `throws` au point (1). Si l'exception `PasDefini` est levée lors de l'appel à factorielle, elle est rattrapée au niveau de la construction `try catch` des points (3) (3 bis) et un message indiquant que la factorielle du nombre entré n'est pas définie est alors affiché. Voici deux exécutions du programme avec des valeurs différentes pour `x` (l'exception est levée puis rattrapée lors de la deuxième exécution) :

```
> java Factorielle
Entrez un nombre (petit):4
24
> java Factorielle
Entrez un nombre (petit):-3
La factorielle de -3 n'est pas définie !
```

Annexe : quelques exceptions prédéfinies

Voici quelques exceptions prédéfinies dans Java :

- `NullPointerException` : utilisation de `length` ou accès à un case d'un tableau valant `null` (c'est à dire non encore créé par un new).
- `ArrayIndexOutOfBoundsException` : accès à une case inexistante dans un tableau.
- `ArrayIndexOutOfBoundsException` : accès au i^{eme} caractère d'un chaîne de caractères de taille inférieure à i.

– `ArrayIndexOutOfBoundsException` : création d'un tableau de taille négative.
– `NumberFormatException` : erreur lors de la conversion d'un chaîne de caractères en nombre.

La classe `Terminal` utilise également l'exception `TerminalException` pour signaler des erreurs.

Chapitre 9

Courte introduction au type String

Il existe en java un type des chaînes de caractères. Ce type s'appelle `String` et les valeurs de ce type sont des objets. Nous verrons plus tard dans le cours ce que sont les objets, comment on les crée, etc. Pour l'instant, nous allons juste présenter très rapidement comment on peut utiliser des chaînes de caractères dans les exercices et exemples que nous traitons.

On peut utiliser le type `String` pour déclarer des variables ou des paramètres de méthodes ou encore des tableaux, exactement comme on utilise les autres types comme `int` ou `boolean`.

Les valeurs constantes du type s'écrivent avec des doubles quotes. Par exemple `"Bonjour"` est une chaîne de caractères. Si l'on veut écrire une chaîne comprennant des doubles quotes, il faut les faire précéder d'un caractère *barre oblique* : `"on m'appelle \"toto\"!"`.

La concaténation de chaînes qui permet de coller une chaîne à la suite d'une autre s'écrit avec l'opérateur `+`. Par exemple `"Bra"+"vo"` est une expression dont la valeur est la chaîne `"Bravo"`.

9.1 Utilisation de méthodes

On peut utiliser des méthodes spécifiques pour les chaînes de caractères au moyen d'une syntaxe nouvelle pour nous. Cette syntaxe c'est : une valeur de type string, un point, le nom de la méthode et les paramètres entre parenthèses. Par exemple, on peut utiliser la méthode `length` (qui n'a aucun paramètre) pour connaître la longueur de la chaîne. En voici un exemple.

Listing 9.1 – (lien vers le code brut)

```
1  public class MethodeString{
2      public static void main(String[] args){
3          String s = "la chaine 1";
4          int x;
5          x = s.length();
6          Terminal.ecrireStringln("La longueur de la chaine \""+s
7                                  + "\" est " + x);
8          x = "une autre chaine".length();
9          Terminal.ecrireStringln("La longueur de l'autre chaine est " + x);
10         x = ("Bra" + "vo").length();
11         Terminal.ecrireStringln("La longueur de Bravo est " + x);
12     }
13 }
```

Voici quelques autres méthodes intéressantes :

– `charAt(int n)` : cette méthode renvoie le nième caractère de la chaîne, la numérotation commence à 0. Par exemple, si `s` est une `String`, `s.charAt(0)` renvoie le premier caractère (type `char`) de `s`.
– `toCharArray()` permet de transformer une chaîne en un tableau de `char`. Par exemple, si `s` est la `String "bonjour"`, `s.toCharArray()` renvoie un tableau de 7 `char` :

0	1	2	3	4	5	6
'b'	'o'	'n'	'j'	'o'	'u'	'r'

– `compareTo(String s2)` : compare deux chaînes selon l'ordre lexicographique (l'ordre du dictionnaire). Si `s1` et `s2` sont deux `String`, `s1.compareTo(s2)` renvoie un entier. Cet entier est négatif si `s1` est plus petit que `s2`, positif si `s1` est plus grand que `s2`, et 0 si `s1` et `s2` sont égales.
– `s1.toLowerCase()` et `s1.toUpperCase()` renvoient une nouvelle chaîne égale à `s1` mais avec toutes les lettres en minuscule et en majuscule respectivement.

Listing 9.2 – (lien vers le code brut)

```
1  public class ExChaine2{
2      public static void main (String [] arguments){
3          String s1 = "bonjour";
4          String s2;
5          Terminal.ecrireString("Entrez une chaine: ");
6          s2 = Terminal.lireString();
7          Terminal.ecrireCharln(s1.charAt(0));
8          Terminal.ecrireStringln(s1.toUpperCase());
9          Terminal.ecrireIntln(s1.compareTo(s2));
10         Terminal.ecrireStringln("" + s1.equals(s2));
11     }
12 }
```

9.2 Egalité de chaînes

La seule chose un peu différente par rapport aux types que nous avons vus jusqu'ici est que les tests de comparaison `==` et `!=` donnent parfois des résultats surprenants quand on les utilise avec des chaînes. Pour comparer des chaînes de caractères, il est préférable d'utiliser la méthode `equals` ou la négation de cette méthode, comme illustré dans l'exemple ci-dessous.

Listing 9.3 – (lien vers le code brut)

```
1  public class ExChaine{
2      public static void main(String[] args){
3          String s1 = "Bonjour";
4          String s2 = "C'est \"bien\" ";
5          String s3;
6          String[] ts = {"Paul", "Andre", "Jacques", "Odette"};
7          Terminal.ecrireStringln(s2);
8          Terminal.ecrireString("Entrez une chaine: ");
9          s3=Terminal.lireString();
10         Terminal.ecrireStringln("s3: " + s3);
11         s2 = "Bon";
12         s3 = s2 + "jour";
13         if (s1 != s3){
```

```
14              Terminal.ecrireStringln("Bizarre:␣s1␣n'est␣pas␣egal␣a␣s3!");
15              Terminal.ecrireStringln("s1:␣" + s1 + ":");
16              Terminal.ecrireStringln("s3:␣" + s3+ ":");
17          }
18          if (s1.equals(s3)){
19              Terminal.ecrireStringln("s1␣est␣quand␣meme␣egal␣a␣s3!");
20          }
21          if (!s1.equals(s3)){
22              Terminal.ecrireStringln("s1␣n'est␣toujours␣pas␣egal␣a␣s3!");
23          }
24      }
25  }
```

Le phénomène que nous observons ici est le même que pour les tableaux : il y a deux notions d'égalité différentes. Une égalité d'identité testée par == et != . Cela permet de savoir si deux variables sont des noms différents pour la même chose (le même tableau, la même chaîne). Une égalité de contenu pour savoir si les deux variables contiennent les mêmes valeurs (les mêmes contenus de cases pour les tableaux, les mêmes caractères dans le même ordre pour les chaînes). Cette égalité de contenu est testée par la méthode `equals`.

9.3 Paramètre de la méthode main

Depuis le début de l'année, nous utilisons systématiquement la méthode `main` avec un paramètre de type `String[]`, c'est à dire un tableau de chaînes de caractères. Ce paramètre permet de transférer des informations entre la ligne de commande et le programme java. Prenons un exemple où le programme se contente d'afficher les valeurs passées sur la ligne de commande.

Listing 9.4 – (lien vers le code brut)

```
1  public class LigneCommande{
2      public static void main(String[] args){
3          for (int i=0; i < args.length; i++){
4              Terminal.ecrireStringln(args[i]);
5          }
6      }
7  }
```

Voici un exemple d'exécution :

```
> java LigneCommande un deux trois
un
deux
trois
```

La tableau `args` dans cette exécution a trois cases. Sa valeur est

0	1	2
"un"	"deux"	"trois"

Notons que même si l'on passe un nombre en paramètre, celui-ci est contenu dans le tableau sous forme d'une chaîne.

```
> java LigneCommande un 12 56 deux
un
```

```
12
56
deux
```

La tableaux args vaut

0	1	2	3
"un"	"12"	"56"	"trois"

Si l'on veut transformer cette chaîne en un entier, il faut utiliser une fonction de conversion.

9.4 Conversion entre chaînes et autres types

Pour les chaînes de caractères, il n'existe pas de conversion explicite avec d'autres types de données.

Par exemple, si l'on essaie d'affecter une valeur de type `String` à une variable de type `int` comme dans l'exemple suivant, cela produit une erreur.

Listing 9.5 – (lien vers le code brut)

```
1 public class StringInt{
2     public static void main(String[] args){
3         int x;
4         String s = "12";
5         x = s;
6     }
7 }
```

A la compilation, on obtient l'erreur suivante :

```
> javac StringInt.java
StringInt.java:5: incompatible types
found   : java.lang.String
required: int
        x = s;
            ^
1 error
```

Pour réaliser la conversion, il faut utiliser la méthode `Integer.parseInt` et lui donner en paramètre la chaîne à convertir.

Listing 9.6 – (lien vers le code brut)

```
1 public class StringInt2{
2     public static void main(String[] args){
3         int x;
4         String s = "12";
5         x = Integer.parseInt(s);
6         Terminal.ecrireIntln(x);
7     }
8 }
```

Pour convertir une valeur de type double, il faut utiliser la méthode `Double.parseDouble` et pour le type boolean, la méthode `Boolean.parseBoolean`.

Pour convertir dans l'autre sens, un int en chaîne, le plus simple est d'utiliser l'opérateur de concaténation : `""+12` (pour les doubles `""+12.3`, pour les booléens `""+true`). On concatène la chaîne vide avec la valeur à convertir.