

# Développement I (Algorithmique et Java)

DEV 1 - DEV

2022-2023

A.HALLAL  
C.LEIGNEL  
DP.BISCHOP  
J.BELEHO  
N.RICHARD  
S.REXHEP  
L.BEECKMANS  
P.HAUWEELE  
D.BOIGELOT



Haute École Bruxelles-Brabant  
École Supérieure d'Informatique  
Bachelor en Informatique

Rue Royale 67 1000 Bruxelles  
+32 (0)2 219 15 46  
esi@he2b.be

Ce syllabus a été écrit initialement par Stefan Monabaliu pour un cours de « Logique et techniques de programmation ». Les grilles de cours ont été revues notamment suite au décret paysage et le cours d’aujourd’hui est un cours de développement qui s’intitule simplement « Développement I ».

Ce syllabus a été adapté par Catherine Leruste, Laurent Beeckmans, Marco Codutti et Pierre Bettens au fil de ses versions.

Document produit avec L<sup>A</sup>T<sub>E</sub>X.

Version du 19 septembre 2022.

## Crédits



Icône « Book » de Thomas Elbig<sup>1</sup> de The noun project<sup>2</sup>



Icône « Maze » de Gilbert Bages<sup>3</sup> de The noun project



Icône « Attention » de Vineet Kumar Thakur<sup>4</sup> de The noun project



Icône « Clipped Postit » de Icon 54<sup>5</sup> de The noun project



Icône « Stop » de Gregor Cresnar<sup>6</sup> de The noun project



Ce document est mis à disposition selon les termes de la Licence Creative Commons Attribution - Partage dans les Mêmes Conditions 4.0 International.

<https://creativecommons.org/licenses/by-sa/4.0/deed.fr>

Les autorisations au-delà du champ de cette licence peuvent être demandées à [esi-dev1-list@he2b.be](mailto:esi-dev1-list@he2b.be).

- 
1. <https://thenounproject.com/dergraph>
  2. <https://thenounproject.com>
  3. <https://thenounproject.com/gilbertbages>
  4. <https://thenounproject.com/vkvineet>
  5. <https://thenounproject.com/icon54app/>
  6. <https://thenounproject.com/grega.cresnar>

# Table des matières

<b>I</b>	<b>Introduction générale</b>	<b>7</b>
1	Résoudre des problèmes	9
1.1	La notion de problème . . . . .	9
1.2	Procédure de résolution . . . . .	11
1.3	Ressources . . . . .	14
2	Les algorithmes et les programmes	15
2.1	Algorithmes et programmes . . . . .	15
2.2	Les phases d'élaboration d'un programme . . . . .	18
2.3	Conclusion . . . . .	20
2.4	Ressources . . . . .	20
<b>II</b>	<b>Les bases de l'algorithmique et de la programmation</b>	<b>21</b>
3	Spécifier le problème	23
3.1	Déterminer les données et le résultat . . . . .	23
3.2	Les noms . . . . .	24
3.3	Les types . . . . .	25
3.4	Résumés . . . . .	27
3.5	Exemples numériques . . . . .	28
4	Premiers algorithmes	29
4.1	Exercice résolu : un problème simple . . . . .	31
4.2	Résolution d'un second problème simple . . . . .	35
4.3	Décomposer les calculs ; variables et assignation . . . . .	36
4.4	Quelques difficultés liées au calcul . . . . .	40
4.5	Des algorithmes et des programmes de qualité . . . . .	47
4.6	Améliorer la lisibilité . . . . .	49
4.7	Appel d'algorithme, appel de méthode . . . . .	53
4.8	Interagir avec l'utilisateur . . . . .	54
5	Premiers programmes	57
5.1	Introduction . . . . .	57
5.2	Environnement de développement . . . . .	61
5.3	La grammaire du langage . . . . .	63
6	Une question de choix	65
6.1	Le si ( <i>if-then</i> ) . . . . .	65
6.2	Le si-sinon ( <i>if-then-else</i> ) . . . . .	69
6.3	Le si-sinon-si . . . . .	72

6.4	Expression booléenne . . . . .	74
6.5	Le selon-que ( <i>switch</i> ) . . . . .	75
<b>7</b>	<b>Module et références</b>	<b>79</b>
7.1	Décomposer le problème . . . . .	79
7.2	Exemple . . . . .	80
7.3	Paramètres et valeur de retour . . . . .	81
7.4	Type primitif et type référence . . . . .	84
<b>8</b>	<b>Un travail répétitif</b>	<b>87</b>
8.1	La notion de travail répétitif . . . . .	87
8.2	Une même instruction, des effets différents . . . . .	88
8.3	Le « tant que » ( <i>while</i> ) . . . . .	90
8.4	Le « pour » ( <i>for</i> ) . . . . .	92
8.5	« faire – tant que » . . . . .	96
8.6	Quel type de boucle choisir ? . . . . .	97
8.7	Exercices résolus . . . . .	98
<b>9</b>	<b>Les tests</b>	<b>105</b>
9.1	Les types d’erreurs et de tests . . . . .	105
9.2	Planifier les tests . . . . .	106
9.3	JUnit . . . . .	107
9.4	Exemple . . . . .	109
<b>III</b>	<b>Les tableaux</b>	<b>111</b>
<b>10</b>	<b>Les tableaux</b>	<b>113</b>
10.1	Utilité des tableaux . . . . .	114
10.2	Définitions . . . . .	117
10.3	Déclaration - création - initialisation . . . . .	117
10.4	Tableau et paramètres . . . . .	120
10.5	Parcours d’un tableau . . . . .	122
10.6	Taille logique et taille physique . . . . .	123
10.7	Les tableaux ont comme premier indice 0 . . . . .	124
<b>11</b>	<b>Algorithmes d’insertions et de recherches</b>	<b>127</b>
11.1	Insertion dans un tableau non trié . . . . .	128
11.2	Insertion dans un tableau trié . . . . .	131
11.3	La recherche dichotomique . . . . .	135
<b>12</b>	<b>Algorithmes de tris</b>	<b>139</b>
12.1	Motivation . . . . .	139
12.2	Tri par insertion . . . . .	141
12.3	Tri par sélection des minima successifs . . . . .	143
12.4	Tri bulle . . . . .	145
12.5	Cas particuliers . . . . .	146
12.6	Références . . . . .	147

<b>IV</b>	<b>Compléments</b>	<b>149</b>
13	<b>Les types</b>	<b>151</b>
13.1	Les types primitifs . . . . .	152
13.2	Les types références . . . . .	160
14	<b>Les chaines et les String</b>	<b>165</b>
14.1	Interface de programmation applicative . . . . .	166
14.2	Chaine et caractère . . . . .	166
14.3	Longueur . . . . .	166
14.4	Le contenu d'une chaine . . . . .	167
14.5	La concaténation . . . . .	167
14.6	Manipuler les caractères . . . . .	168
14.7	L'alphabet . . . . .	170
14.8	Chaine et nombre . . . . .	171
14.9	Extraction de sous-chaines . . . . .	172
14.10	Recherche de sous-chaine . . . . .	172
15	<b>Les exceptions</b>	<b>175</b>
15.1	Motivation . . . . .	175
15.2	L'instruction <code>try-catch</code> . . . . .	177
15.3	L'instruction <code>throw</code> . . . . .	178
16	<b>La Javadoc</b>	<b>181</b>
16.1	La documentation . . . . .	182
16.2	La documentation dans le code . . . . .	182
16.3	Le commentaire <code>javadoc</code> . . . . .	183
16.4	L'outil <code>javadoc</code> . . . . .	187
<b>V</b>	<b>Annexes</b>	<b>189</b>
<b>A</b>	<b>Exercices</b>	<b>191</b>
A.1	Exercices : premiers algorithmes et programmes . . . . .	192
A.2	Exercices : tracer un algorithme . . . . .	194
A.3	Exercices : division entière, quotient et reste . . . . .	196
A.4	Exercices : structures alternatives . . . . .	199
A.5	Exercices : modules et méthodes . . . . .	203
A.6	Exercices : structures itératives . . . . .	205
A.7	Exercices : tableaux . . . . .	211
A.8	Exercices : chaine et String . . . . .	216
A.9	Exercices récapitulatifs . . . . .	217
<b>B</b>	<b>Exercices résolus</b>	<b>223</b>
B.1	Exercice résolu : le blackjack . . . . .	224
<b>C</b>	<b>Les fiches</b>	<b>229</b>
	Un calcul simple . . . . .	230
	Un calcul complexe . . . . .	232
	Un nombre pair . . . . .	234
	Maximum de deux nombres . . . . .	236

Parcours complet d'un tableau . . . . .	239
Maximum dans un tableau . . . . .	241
Parcours partiel d'un tableau . . . . .	244
Tableau non trié . . . . .	247
Tableau trié . . . . .	251
Recherche dichotomique . . . . .	257
Tri d'un tableau . . . . .	259
<b>D Bonnes pratiques</b>	<b>263</b>
D.1 Bonnes pratiques générales . . . . .	263
D.2 Tester un booléen . . . . .	264
D.3 Assigner un booléen . . . . .	265
D.4 Assigner une valeur en fonction d'une condition . . . . .	265
D.5 Interrompre une boucle <code>for</code> . . . . .	266
D.6 Plusieurs <code>return</code> . . . . .	267
<b>Index</b>	<b>271</b>

## Première partie

# Introduction générale





# Chapitre 1

## Résoudre des problèmes

« *L’algorithmique est le permis de conduire de l’informatique. Sans elle, il n’est pas concevable d’exploiter sans risque un ordinateur.* »<sup>7</sup>

Ce chapitre a pour but de vous faire comprendre ce qu’est une *procédure de résolution de problèmes*.

### Contenu

1.1	La notion de problème . . . . .	<b>9</b>
1.1.1	Préliminaires : utilité de l’ordinateur . . . . .	9
1.1.2	Poser le problème . . . . .	10
1.2	Procédure de résolution . . . . .	<b>11</b>
1.2.1	Chronologie des opérations . . . . .	11
1.2.2	Les opérations élémentaires . . . . .	11
1.2.3	Les opérations bien définies . . . . .	12
1.2.4	Opérations soumises à une condition . . . . .	13
1.2.5	Opérations à répéter . . . . .	13
1.2.6	À propos des données . . . . .	13
1.2.7	Conclusion . . . . .	14
1.3	Ressources . . . . .	<b>14</b>

## 1.1 La notion de problème

### 1.1.1 Préliminaires : utilité de l’ordinateur

L’ordinateur est une machine. Mais une machine intéressante dans la mesure où elle est destinée d’une part, à nous décharger d’une multitude de tâches peu valorisantes, rébarbatives telles que le travail administratif répétitif, mais surtout parce qu’elle est capable de nous aider, voire nous remplacer, dans des tâches plus ardues qu’il

<sup>7</sup>. [CORMEN e.a., Algorithmique, Paris, Edit. Dunod, 2010, (Cours, exercices et problèmes), p. V]

nous serait impossible de résoudre sans son existence (conquête spatiale, prévision météorologique, jeux vidéo...).

En première approximation, nous pourrions dire que l'ordinateur est destiné à nous remplacer, à faire à notre place (plus rapidement et probablement avec moins d'erreurs) un travail nécessaire à la résolution de **problèmes** auxquels nous devons faire face. Attention ! Il s'agit bien de résoudre des *problèmes* et non des mystères (celui de l'existence, par exemple). Il faut que la question à laquelle nous souhaitons répondre soit **accessible à la raison**.

### 1.1.2 Poser le problème

Un préalable à l'activité de résolution d'un problème est de bien **définir** d'abord quel est le problème posé, en quoi il consiste exactement ; par exemple, faire un baba au rhum, réussir une année d'études, résoudre une équation mathématique...

Un problème bien posé doit mentionner l'**objectif à atteindre**, c'est-à-dire la situation d'arrivée, le but escompté, le résultat attendu. Généralement, tout problème se définit d'abord explicitement par ce que l'on souhaite obtenir.

La formulation d'un problème ne serait pas complète sans la connaissance **du cadre dans lequel le problème est posé** : de quoi dispose-t-on, quelles sont les hypothèses de base, quelle est la situation de départ ? Faire un baba au rhum est un problème tout à fait différent s'il faut le faire en plein désert ou dans une cuisine super équipée ! D'ailleurs, dans certains cas, la première phase de la résolution d'un problème consiste à identifier et mettre à sa disposition les éléments nécessaires à sa résolution : dans notre exemple, ce serait se procurer les ingrédients et les ustensiles de cuisine.

Un problème ne sera véritablement bien spécifié que s'il s'inscrit dans le schéma suivant :

**étant donné** [la situation de départ] **on demande** [l'objectif]

Parfois, la première étape dans la résolution d'un problème est de préciser ce problème à partir d'un énoncé flou : il ne s'agit pas nécessairement d'un travail facile !

**Exercice** Un problème flou.

Soit le problème suivant : « Déterminer le meilleur chemin pour aller de Anvers à Charleroi en passant par Bruxelles. ».

Ce problème est-il bien posé ?

Expliquez pourquoi cet énoncé n'est pas bon.

Proposez un énoncé qui soit acceptable.

Une fois le problème correctement posé, on passe à la recherche et à la description d'une **méthode/procédure de résolution**, afin de savoir comment faire pour atteindre l'objectif demandé à partir de ce qui est donné. Le **nom** donné à une méthode de résolution varie en fonction du cadre dans lequel se pose le problème : *façon de procéder, mode d'emploi, marche à suivre, guide, patron, modèle, recette de cuisine, méthode ou plan de travail, algorithme mathématique, programme, directives d'utilisation...*

## 1.2 Procédure de résolution

Une **procédure de résolution** est une description en termes compréhensibles par l'exécutant de la **marche à suivre** pour résoudre un problème donné.

On trouve beaucoup d'exemples dans la vie courante : recette de cuisine, mode d'emploi d'un GSM, description d'un itinéraire, plan de montage d'un jeu de construction, etc. Il est clair qu'il y a une infinité de rédactions possibles de ces différentes marches à suivre. Certaines pourraient être plus précises que d'autres, d'autres par contre pourraient s'avérer exagérément explicatives.

Des différents exemples de procédures de résolution se dégagent les caractéristiques suivantes :

- ▷ toutes ont un **nom** ;
- ▷ elles s'expriment dans un **langage** (français, anglais, dessins. . .) ;
- ▷ l'ensemble de la procédure consiste en une **série chronologique** d'instructions ou de phrases (parfois numérotées) ;
- ▷ une instruction se caractérise par un ordre, une action à accomplir, une **opération** à exécuter sur les **données** du problème ;
- ▷ certaines phrases justifient ou expliquent ce qui se passe : ce sont des **commentaires**.

On pourra donc définir, en première approximation, une procédure de résolution comme un texte, écrit dans un certain langage, qui décrit une suite d'actions à exécuter dans un ordre précis, ces actions opérant sur des objets issus des données du problème.

Traduite en termes informatiques, une telle procédure d'exécutions d'actions dans un contexte précis, sera appelée un **algorithme**. Nous y reviendrons et le définirons tout à fait précisément plus loin, dans notre contexte.

### 1.2.1 Chronologie des opérations

Pour ce qui concerne l'ordinateur, le travail d'exécution d'une marche à suivre est impérativement **séquentiel**. C'est-à-dire que les instructions d'une procédure de résolution sont exécutées **une et une seule fois** dans l'ordre où elles apparaissent. Cependant certains artifices d'écriture permettent de **répéter** l'exécution d'opérations ou de la **conditionner** (c'est-à-dire de choisir si l'exécution aura lieu ou non en fonction de la réalisation d'une condition).

### 1.2.2 Les opérations élémentaires

Dans la description d'une marche à suivre, la plupart des opérations sont introduites par un **verbe** (*remplir, verser, prendre, peler*, etc.). L'exécutant ne pourra exécuter une action que s'il la comprend : cette action doit, pour lui, être une action élémentaire, une action qu'il peut réaliser sans qu'on ne doive lui donner des explications complémentaires. Ce genre d'opération élémentaire est appelée **primitive**.

Ce concept est évidemment relatif à ce qu'un exécutant est capable de réaliser. Cette capacité, il la possède d'abord parce qu'il est **construit** d'une certaine façon (capacité innée). Ensuite parce que, par construction aussi, il est doté d'une faculté

d'**apprentissage** lui permettant d'assimiler, petit à petit, des procédures non élémentaires qu'il exécute souvent. Une opération non élémentaire pourra devenir une primitive un peu plus tard.

### 1.2.3 Les opérations bien définies

Il arrive de trouver dans certaines marches à suivre des opérations qui peuvent dépendre d'une certaine manière de l'appréciation de l'exécutant. Par exemple, dans une recette de cuisine nous pourrions lire : *ajouter **un peu** de vinaigre, saler et poivrer **à volonté**, laisser cuire une **bonne** heure dans un four **bien** chaud, etc.*

Des instructions floues de ce genre sont dangereuses à faire figurer dans une bonne marche à suivre car elles font appel à une appréciation arbitraire de l'exécutant. Le résultat obtenu risque d'être imprévisible d'une exécution à l'autre. De plus, les termes du type *environ, beaucoup, pas trop* et *à peu près* sont intraduisibles et proscrites au niveau d'un langage informatique!<sup>8</sup>

Une **opération bien définie** est donc une opération débarrassée de tout vocabulaire flou et dont le résultat est **entièrement prévisible**. Des versions « bien définies » des exemples ci-dessus pourraient être : *ajouter 2 cl de vinaigre, ajouter 5 g de sel et 1 g de poivre, laisser cuire 65 minutes dans un four chauffé à 220 ° C, etc.*

Afin de mettre en évidence la difficulté d'écrire une marche à suivre claire et non ambiguë, l'expérience suivante a été menée et vous pouvez la reproduire.

**Expérience.** Le dessin.

Cette expérience s'effectue en groupe. Le but est de faire un dessin et de permettre à une autre personne, qui ne l'a pas vu, de le reproduire fidèlement, au travers d'une « marche à suivre ».

1. Chaque personne prend une feuille de papier et y dessine quelque chose en quelques traits précis. Le dessin ne doit pas être trop compliqué ; on ne teste pas ici vos talents de dessinateur ! (ça peut être une maison, une voiture. . .)
2. Sur une **autre** feuille de papier, chacun rédige des instructions permettant de reproduire fidèlement son propre dessin. Attention ! Il est important de ne **jamais faire référence à la signification du dessin**. Ainsi, on peut écrire : « dessine un cercle » mais certainement pas : « dessine une roue ».
3. Chacun cache à présent son propre dessin et échange sa feuille d'instructions avec celle de quelqu'un d'autre.
4. Chacun s'efforce ensuite de reproduire le dessin d'un autre en suivant **scrupuleusement** les instructions indiquées sur la feuille reçue en échange, **sans tenter d'initiative** (par exemple en croyant avoir compris ce qu'il faut dessiner).

---

8. Toute personne intéressée découvrira dans la littérature spécialisée que même les procédures de génération de nombres aléatoires sont elles aussi issues d'algorithmes mathématiques tout à fait déterministes. Des algorithmes dont la sortie ne dépend que des paramètres, c'est-à-dire qui, étant donné des entrées données, fournira systématiquement la même sortie.

5. Nous examinerons enfin les différences entre l'original et la reproduction et nous tenterons de comprendre pourquoi elles se sont produites (par imprécision des instructions ou par mauvaise interprétation de celles-ci par le dessinateur...)

Quelles réflexions cette expérience vous inspire-t-elle ? Quelle analogie voyez-vous avec une marche à suivre donnée à un ordinateur ?



Dans cette expérience, nous imposons que la « marche à suivre » ne mentionne aucun mot expliquant le sens du dessin (mettre « rond » et pas « roue » par exemple). Pourquoi, à votre avis, avons-nous imposé cette contrainte ?

#### 1.2.4 Opérations soumises à une condition

En français, l'utilisation de conjonctions ou locutions conjonctives du type *si*, *selon que*, *au cas où*... présuppose la possibilité de ne pas exécuter certaines opérations en fonction de certains événements. D'une fois à l'autre, certaines de ses parties seront ou non exécutées.

**Exemple :** Si la viande est surgelée, la décongeler en la déposant dans une assiette le matin.

#### 1.2.5 Opérations à répéter

De la même manière, il est possible d'exprimer en français une exécution répétitive d'opérations en utilisant les mots *tous*, *chaque*, *tant que*, *jusqu'à ce que*, *chaque fois que*, *aussi longtemps que*, *faire x fois*...

Dans certains cas, le nombre de répétitions est connu à l'avance (*répéter 10 fois*) ou déterminé par une durée (*faire cuire pendant 30 minutes*) et dans d'autres cas il est inconnu. Dans ce cas, la fin de la période de répétition d'un bloc d'opérations dépend alors de la réalisation d'une condition (*lancer le dé jusqu'à ce qu'il tombe sur 6*, *faire chauffer jusqu'à évaporation complète*...).

En informatique, lorsqu'il y a répétition organisée, on parle de **boucle**. On en retrouve beaucoup, sous différentes formes, dans les codes informatiques.

L'exemple ci-dessous permet d'illustrer le danger de boucle infinie, due à une mauvaise formulation de la condition d'arrêt. Si l'on demande de *lancer le dé — de 6 faces — jusqu'à ce que la valeur obtenue soit 7*, une personne dotée d'intelligence comprend que la condition est impossible à réaliser, mais un robot appliquant cette directive à la lettre lancera le dé éternellement.

#### 1.2.6 À propos des données

Les types d'objets figurant dans les diverses procédures de résolution sont fonction du cadre dans lequel s'inscrivent ces procédures, du domaine d'application de ces marches à suivre. Par exemple, pour une recette de cuisine, ce sont les ingrédients. Pour un jeu de construction ce sont les briques.

L'ordinateur, quant à lui, manipule principalement des données numériques et textuelles. Nous verrons plus tard comment on peut combiner ces données élémentaires pour obtenir des données plus complexes.

### 1.2.7 Conclusion

En conclusion des points précédents, lorsque vous écrivez procédure de résolution, vérifiez votre check-list !

- ☐ Le problème est-il bien posé ?
- ☐ L'ordre des opérations de ma procédure est-il indiqué et correct ?
- ☐ Ma procédure est-elle constituée d'opérations élémentaires ?
- ☐ Les opérations utilisées sont-elles bien définies ? Pas de flou ?
- ☐ Les opérations soumises à une condition sont-elles bien identifiées ?
- ☐ Les opérations à répéter sont-elles bien identifiées ?
- ☐ Les données manipulées sont-elles clairement indiquées ?

et bien sûr :

- ☐ Ma procédure résout-elle effectivement le problème posé ?

## 1.3 Ressources

Pour prolonger votre réflexion sur le concept d'algorithme nous vous proposons quelques ressources en ligne :

- ▷ Les Sépas 18 - Les algorithmes<sup>9</sup>
- ▷ Les Sépas 11 - Un bug<sup>10</sup>
- ▷ Le crépier psycho-rigide comme algorithme<sup>11</sup>
- ▷ Le baseball multicolore comme algorithme<sup>12</sup>
- ▷ Le jeu de Nim comme algorithme<sup>13</sup>
- ▷ Code Studio<sup>14</sup>, un site d'apprentissage et d'initiation à l'algorithmique et la notion de programme.
- ▷ Coding game<sup>15</sup>, un site d'apprentissage de la programmation un peu différent du précédent... et un peu plus complexe.

---

9. <https://www.youtube.com/watch?v=hG9Jty7P6Es>

10. <https://www.youtube.com/watch?v=deIOGV5sWTY>

11. <https://pixees.fr/?p=446>

12. <https://pixees.fr/?p=450>

13. <https://pixees.fr/?p=443>

14. <http://studio.code.org/>

15. <https://www.codingame.com>

# Chapitre 2

## Les algorithmes et les programmes

Notre but étant de faire de l'informatique, il convient de restreindre notre étude à des notions plus précises, plus spécialisées, gravitant autour de la notion de *traitement automatique de l'information*. Voyons ce que cela signifie.

### Contenu

2.1	Algorithmes et programmes . . . . .	<b>15</b>
2.1.1	Algorithme . . . . .	15
2.1.2	Programme . . . . .	16
2.1.3	Les constituants principaux de l'ordinateur . . . . .	17
2.1.4	Exécution d'un programme . . . . .	18
2.2	Les phases d'élaboration d'un programme . . . . .	<b>18</b>
2.3	Conclusion . . . . .	<b>20</b>
2.4	Ressources . . . . .	<b>20</b>

## 2.1 Algorithmes et programmes

Décrivons la différence entre un algorithme et un programme et comment un ordinateur peut exécuter un programme.

### 2.1.1 Algorithme

Un algorithme appartient au vaste ensemble des *marches à suivre*, et plus spécifiquement les *procédures de résolution*.

**Algorithme : Procédure de résolution** d'un problème contenant des opérations bien définies portant sur des informations, s'exprimant dans une séquence définie sans ambiguïté, destinée à être traduite dans un langage de programmation.



Comme toute marche à suivre, un algorithme doit s'exprimer dans un certain langage — *a priori* le langage naturel (le français) — mais il y a d'autres possibilités : organigramme, arbre programmatique, pseudocode...

L'algorithme est destiné à être compris par une personne et à être traduit — ce qui n'est pas immédiat — en un programme.

### Exemple simple

Calculer la circonférence d'un cercle.

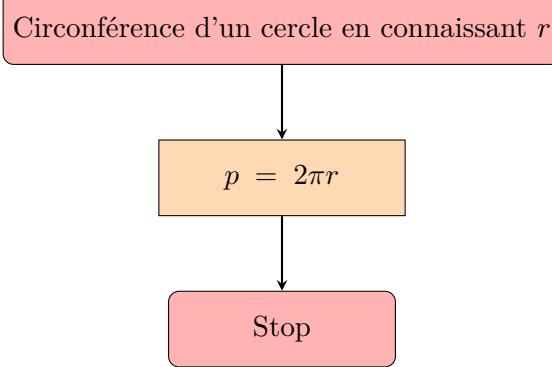
Un algorithme en **langage naturel** pourrait être :

En connaissant le rayon, le calcul de la circonférence se fait grâce à la formule

$$p = 2\pi r$$

*langage naturel*

En utilisant un **organigramme** l'algorithme peut se décrire comme suit (et nous y reviendrons plus longuement plus tard) :



ORGANIGRAMME 1 – Illustration d'un organigramme

En **pseudo-code** cette fois, l'algorithme aurait l'allure suivante :

```

algorithm circleCircumference(radius : real) → real
  return 2 π radius
  
```

*pseudocode*

Pour un ordinateur, il faudra du **code** pour obtenir un **programme**. Nous en parlons ci-dessous.

#### 2.1.2 Programme



Un **programme** n'est rien d'autre que la représentation d'un algorithme dans un langage compris par un ordinateur (par exemple : Assembleur, Cobol, Java, C++, Python...). Ce type de langage est appelé **langage de programmation**.

Écrire un programme correct suppose donc la parfaite connaissance du langage de programmation et de sa **syntaxe**, qui est en quelque sorte la grammaire du langage. Mais ce n'est pas suffisant ! Puisque le programme est la représentation d'un algorithme, il faut que celui-ci soit correct pour que le programme le soit. Un programme correct résulte donc d'une démarche logique correcte (algorithme correct) et de la connaissance de la syntaxe d'un langage de programmation. La syntaxe d'un langage est – très – précise.



Il est donc indispensable d'élaborer des algorithmes corrects avant d'espérer concevoir des programmes corrects.

### Un exemple simple (suite)

En Java, le programme aurait l'allure suivante mais ne serait pas utilisable en l'état. Il lui manque une méthode principale. Nous y reviendrons :

```
1 public class CircleCircumference{  
    public static double circleCircumference(double radius){  
        return 2 * Math.PI * radius;  
4    }  
}
```

java

### 2.1.3 Les constituants principaux de l'ordinateur

Cette section ne fait pas partie de la matière évaluée dans la partie théorique.

Les constituants d'un ordinateur se divisent en **hardware** (matériel) et **software** (logiciel).

Le **hardware** est constitué de l'ordinateur proprement dit et regroupe les entités suivantes :

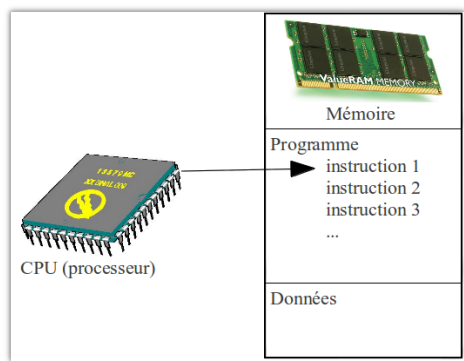
- ▷ **l'organe de contrôle** : c'est le cerveau de l'ordinateur. Il est l'organisateur, le contrôleur suprême de l'ensemble. Il assume l'enchaînement des opérations élémentaires. Il s'occupe également d'organiser l'exécution effective de ces opérations élémentaires reprises dans les programmes.
- ▷ **l'organe de calcul** : c'est le calculateur où ont lieu les opérations arithmétiques ou logiques. Avec l'organe de contrôle, il constitue le **processeur** ou **unité centrale**.
- ▷ **la mémoire centrale** : dispositif permettant de mémoriser, pendant le temps nécessaire à l'exécution, les programmes et certaines données pour ces programmes.
- ▷ **les unités d'échange avec l'extérieur** : dispositifs permettant à l'ordinateur de recevoir des informations de l'extérieur (unités de lecture telles que clavier, souris, écran tactile...) ou de communiquer des informations vers l'extérieur (unités d'écriture telles que écran, imprimantes, signaux sonores...).
- ▷ **les unités de conservation à long terme** : ce sont les mémoires auxiliaires (disques durs, CD ou DVD de données, clés USB...) sur lesquelles sont conservées les procédures (programmes) ou les informations résidentes dont le volume (élevé) ou la fréquence d'utilisation (faible) ne justifient pas la conservation permanente en mémoire centrale. C'est également là que sont conservées les données utilisateur destinées à persister entre les redémarrage de la machine.

Le **software** et plus précisément le système d'exploitation, est l'ensemble des procédures (programmes) s'occupant de la gestion du fonctionnement d'un système informatique et de la gestion de l'ensemble des ressources de ce système (le matériel – les programmes – les données). Il contient notamment des logiciels de traduction

permettant d'obtenir un programme écrit en langage machine (langage technique qui est le seul que l'ordinateur peut comprendre directement, c'est-à-dire exécuter) à partir d'un programme écrit en langage de programmation plus ou moins « évolué » (c'est-à-dire plus ou moins proche du langage naturel).

#### 2.1.4 Exécution d'un programme

Isolons (en les simplifiant) deux constituants essentiels de l'ordinateur afin de comprendre ce qu'il se passe quand un ordinateur exécute un programme. D'une part, la mémoire contient le programme et les données manipulées par ce programme. D'autre part, le processeur va « exécuter » ce programme.



#### Comment fonctionne le processeur ?

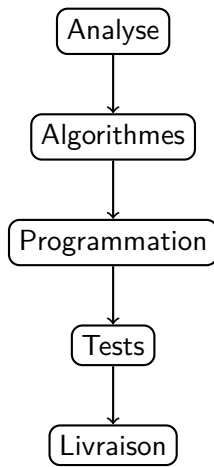
De façon très simplifiée, les étapes suivantes ont lieu :

1. Le processeur lit l'instruction courante.
2. Il exécute cette instruction. Cela peut amener à manipuler les données.
3. L'instruction suivante devient l'instruction courante.
4. On revient au point 1.

Nous voyons bien qu'il s'agit d'un travail automatique ne laissant aucune place à l'initiative !

## 2.2 Les phases d'élaboration d'un programme

Voyons pour résumer un schéma **simplifié** des différentes phases nécessaires au développement d'un programme.



- ▷ Lors de **l'analyse**, le problème doit être compris et clairement précisé. Vous aborderez cette phase plus en détails dans les cours d'analyse.
- ▷ Une fois le problème analysé, et avant de passer à la phase de programmation, il faut réfléchir à ou aux **algorithmes** qui vont permettre de résoudre le problème.
- ▷ On peut alors **programmer** cet algorithme dans le langage de programmation choisi ; Java, Cobol, Assembleur, Javascript, Python ...
- ▷ Vient ensuite la phase de **tests** qui ne manquera pas de montrer qu'il subsiste des problèmes qu'il faut encore corriger. (Vous aurez maintes fois l'occasion de vous en rendre compte lors des séances de laboratoire)
- ▷ Le produit sans bug (connu) peut être **mis en application** ou **livré** à la personne qui vous en a passé la commande.

Notons que ce processus n'est pas linéaire. À chaque phase, on pourra détecter des erreurs, imprécisions ou oublis des phases précédentes et revenir en arrière. Un problème peut être amélioré, complexifié, modifié ou étendu.

**Exemple** Imaginons que je veuille préparer un repas pour mes invités et invitées. Je désire leur proposer une lasagne et décide de suivre une recette. À la dernière minute, j'apprend que l'une des personnes invitées est allergique au lait. Plusieurs choix s'offrent à moi :

- ▷ je modifie ma recette et remplace le lait par du lait de soja ;
- ▷ je change de plat ;
- ▷ je notifie qu'il ou elle n'est plus bienvenue ou qu'elle recevra une tartine<sup>16</sup>.

### Pourquoi passer par la phase « algorithmique » et ne pas directement passer à la programmation ?

Voilà une question que vous ne manquerez pas de vous poser pendant votre apprentissage cette année. Apportons quelques éléments de réflexion.

- ▷ Passer par une phase « algorithmique » permet de séparer deux difficultés : quelle est la marche à suivre ? Et comment l'exprimer dans le langage de programmation choisi ? Pour écrire nos algorithmes nous avons la souplesse de notre sont plus vastes que simplement se restreindre au langage Java (où il faut être précis au « ; » près). Nous commencerons simplement par écrire en langage naturel.
- ▷ De plus, un algorithme écrit facilite le dialogue dans une équipe de développement. « J'ai écrit un algorithme pour résoudre le problème qui nous occupe. Qu'en pensez-vous ? Pensez-vous qu'il est correct ? Avez-vous une meilleure idée ? ». Pour ces discussions, il faut un descriptif de l'algorithme aussi clair que

16. Je m'autorise un accord de proximité.

possible, sans s'encombrer des lourdeurs inhérentes à l'utilisation d'un « vrai » langage de programmation.

- ▷ Enfin, si l'algorithme est écrit, il pourra facilement être traduit dans n'importe quel langage de programmation. La traduction d'un langage de programmation à un autre est un peu moins facile à cause des particularités propres à chaque langage.

Bien sûr, cela n'a de sens que si le problème présente une réelle difficulté algorithmique. Certains problèmes (en pratique, certaines parties de problèmes) sont suffisamment simples pour être directement programmés. Mais qu'est-ce qu'un problème simple ? Cela va évidemment changer tout au long de votre apprentissage. Un problème qui vous paraîtra difficile en début d'année vous paraîtra (enfin, il faut l'espérer !) une évidence en fin d'année.

## 2.3 Conclusion

L'informatisation de problèmes est un processus essentiellement dynamique, contenant des allées et venues constantes entre les différentes étapes. Codifier un algorithme dans un langage de programmation quelconque n'est certainement pas la phase la plus difficile de ce processus. Par contre, élaborer une démarche logique de résolution d'un problème est probablement plus complexe.

Le but de ce premier cours de **développement** est de mêler l'apprentissage des algorithmes et du langage Java comme premier langage.

Nous tenterons :

- ▷ de définir une bonne démarche d'élaboration d'algorithmes (apprentissage de la **logique** de programmation) ;
- ▷ comprendre et apprendre les algorithmes classiques qui ont fait leurs preuves. Pouvoir les utiliser en les adaptant pour résoudre nos problèmes concrets.
- ▷ traduire ces algorithmes en langage Java et les « faire tourner », c'est-à-dire les implémenter sur une machine ; éditer le code, le compiler, l'exécuter...

Le tout devrait avoir pour résultat l'élaboration de *bons programmes*, c'est-à-dire *des programmes dont il est facile de se persuader qu'ils sont corrects* et des programmes dont la maintenance est la plus aisée possible. Dans ce sens, ce cours est bien un premier cours de développement.

Les matières non traitées dans cette première approche du développement le seront, d'abord dans le cours de Développement II (DEV2) et ensuite, Développement III, IV... en fonction du cursus.

## 2.4 Ressources

Pour prolonger votre réflexion sur les notions vues dans ce chapitre, nous vous proposons quelques ressources en ligne :

- ▷ C'est pas sorcier ! Ordinateur, tout un programme <sup>17</sup>

---

17. <https://www.youtube.com/watch?v=c96KP5jZVYk>

## Deuxième partie

# Les bases de l'algorithmique et de la programmation



# Chapitre 3

## Spécifier le problème

Comme nous l'avons dit, un problème ne sera véritablement bien spécifié que s'il s'inscrit dans le schéma suivant :

**étant donné** [les données] **on demande** [résultat]

La première étape dans la résolution d'un problème est de préciser ce problème à partir de l'énoncé, c-à-d de déterminer et préciser les données et le résultat.

### Contenu

3.1	Déterminer les données et le résultat . . . . .	<b>23</b>
3.2	Les noms . . . . .	<b>24</b>
3.3	Les types . . . . .	<b>25</b>
3.3.1	Il n'y a pas d'unité . . . . .	26
3.3.2	Préciser les valeurs possibles . . . . .	26
3.3.3	Le type des données complexes . . . . .	27
3.3.4	Exercice . . . . .	27
3.4	Résumés . . . . .	<b>27</b>
3.4.1	Résumé graphique . . . . .	27
3.4.2	Résumé textuel . . . . .	28
3.5	Exemples numériques . . . . .	<b>28</b>

### 3.1 Déterminer les données et le résultat

La toute première étape est de parvenir à extraire d'un énoncé de problème, quelles sont les données et quel est le résultat attendu<sup>18</sup>. Dans la suite, nous utiliserons un exemple très simple afin d'illustrer les concepts qui nous intéressent.

<sup>18</sup>. Plaçons-nous pour le moment dans le cadre de problèmes où il y a exactement un résultat.

**Exemple.** Soit l'énoncé suivant : « Calculer la surface d'un rectangle à partir de sa longueur et sa largeur ».

Quelles sont les données ? Il y en a deux :

- ▷ la longueur du rectangle ;
- ▷ sa largeur.

Quel est le résultat attendu ? la surface du rectangle.

## 3.2 Les noms

Pour identifier clairement chaque **donnée** et pouvoir y faire référence dans le futur algorithme et dans le programme nous devons lui attribuer un **nom**<sup>19</sup>. Il est important de bien choisir les noms. Le but est de trouver un nom qui soit suffisamment court, tout en restant explicite et ne prêtant pas à confusion.

**Exemple.** Quel nom choisir pour la longueur d'un rectangle ?

On peut envisager les noms suivants :

- ▷ `length` est probablement le plus approprié.
- ▷ `rectangleLength` peut se justifier pour éviter toute ambiguïté avec une autre longueur.
- ▷ `len` peut être admis si le contexte permet de comprendre immédiatement l'abréviation.
- ▷ `l` est à proscrire car pas assez explicite.
- ▷ `theLengthOfMyRectangle` est inutilement long.
- ▷ `foo` (truc en anglais) ou `tmp` ne sont pas de bons choix car ils n'ont aucun lien avec la donnée.

Les noms donnés à chaque donnée seront directement associés à une **variable** lorsque l'algorithme sera traduit en un programme dans un langage de programmation.



**Variable :** Emplacement mémoire nommé pouvant contenir une valeur. Cette valeur peut être remplacée par une autre — elle est variable — au fil de l'exécution du programme.

Nous allons également donner un **nom à l'algorithme** de résolution du problème. Cela permettra d'y faire référence dans les explications mais également de l'utiliser dans d'autres algorithmes. Généralement, un nom d'algorithme est :

- ▷ soit un verbe indiquant ce que fait l'algorithme ;
- ▷ soit un nom indiquant le résultat fourni.

<sup>19</sup>. Dans ces notes, nous nous efforcerons de choisir des noms en anglais, mais vous pouvez très bien choisir des noms français si vous ne vous sentez pas encore suffisamment à l'aise avec l'anglais.



**Exemple.** Quel nom choisir pour l'algorithme qui calcule la surface d'un rectangle ?

On peut envisager le verbe `computeRectangleArea` ou le nom `rectangleArea` (notre préféré). On pourrait aussi simplifier en `area` s'il est évident que le problème traite des rectangles.

Les noms donnés aux algorithmes deviendront généralement les noms des programmes lorsque ces algorithmes seront implémentés.

Notons que les langages de programmation imposent certaines limitations (parfois différentes d'un langage à l'autre) ce qui peut nécessiter une modification du nom lors de la traduction de l'algorithme en un programme. À chaque langage de programmation sont également associées des **conventions d'écriture** que les développeurs respectent. Bien que nous y reviendrons plus en détail dans la section 4.6, notons déjà que la simple convention de nom de méthode change d'un langage à l'autre.

Par exemple pour *rectangle area*, nous utiliserons `rectangleArea` en langage Java mais `rectangle_area` en langage C, C++ et Python.

### 3.3 Les types

Nous allons également attribuer un **type** à chaque donnée ainsi qu'au résultat. Le **type** décrit la nature de son contenu, quelles valeurs elle peut prendre.

Certains langages imposent et vérifient le type de chaque variable tandis que d'autres non. En Java, chaque variable et chaque donnée ont un type.

Dans un premier temps, nous utiliserons les **types** ci-dessous<sup>20</sup>. Une description plus précise des types disponibles en Java est donnée dans la section 13 page 151.

Type	
<code>int</code>	pour les nombres entiers
<code>double</code>	pour les nombres réels
<code>String</code> <sup>21</sup>	pour les chaînes de caractères, les textes par exemple : " <code>Bonjour</code> ", " <code>Bonjour le monde !</code> ", " <code>a</code> ", " <code>"...</code>
<code>boolean</code>	quand la valeur ne peut être que <code>true</code> (vrai) ou <code>false</code> (faux)

<sup>20</sup>. Écrire ces différents types en français n'est pas une faute. Dans ces notes nous utiliserons l'anglais

<sup>21</sup>. Notez la majuscule, indispensable en Java

**Exemples.**

- ▷ Pour la longueur, la largeur et la surface d'un rectangle, on prendra un réel (le type `double`).
- ▷ Pour le nom d'une personne, on choisira une chaîne de caractère (le type `String`).
- ▷ Pour l'âge d'une personne, un entier est indiqué (`int`).
- ▷ Pour décrire si un étudiant est doubleur ou pas, un booléen (`boolean`) est adapté.
- ▷ Pour représenter un mois, on préférera souvent un entier donnant le numéro du mois (par ex : 3 pour le mois de mars) plutôt qu'une chaîne (par ex : "mars") car les manipulations, les calculs seront plus simples.

**3.3.1 Il n'y a pas d'unité**

Un type numérique indique que les valeurs possibles seront des nombres. Il n'y a là aucune notion d'unité. Ainsi, la longueur d'un rectangle, un réel, peut valoir 2.5 mais certainement pas 2.5 *cm*. Si cette unité a de l'importance, il faut la spécifier dans le nom de la donnée ou en commentaire.

**Exemple.** Faut-il préciser les unités pour les dimensions d'un rectangle ?

Si la longueur d'un rectangle vaut 6, on ne peut pas dire s'il s'agit de centimètres, de mètres ou encore de kilomètres. Pour notre problème de calcul de la surface, ce n'est pas important ; la surface n'aura pas d'unité non plus.

Notre seule contrainte est que la longueur et la largeur soient exprimées dans la même unité.

Si, par contre, il est important de préciser que la longueur est donnée en centimètres, on pourrait l'expliciter en la nommant `lengthCm`.

**3.3.2 Préciser les valeurs possibles**

Nous aurions pu introduire un seul type numérique mais nous avons choisi de distinguer les entiers et les réels. Pourquoi ? Préciser qu'une donnée ne peut prendre que des valeurs entières (par exemple dans le cas d'un numéro de mois) aide le lecteur à mieux la comprendre. Nous allons aussi pouvoir définir des opérations propres aux entiers (le reste d'une division par exemple). Enfin, pour des raisons techniques, beaucoup de langages font cette distinction.

Même ainsi, le type choisi n'est pas toujours assez précis. Souvent, la donnée ne pourra prendre que certaines valeurs.

**Exemples.**

- ▷ Un âge est un entier qui ne peut pas être négatif.
- ▷ Un mois est un entier compris entre 1 et 12.

Ces précisions pourront être données en commentaire pour aider à mieux comprendre le problème et sa solution.

**3.3.3 Le type des données complexes**

Parfois, aucun des types disponibles ne permet de représenter la donnée. Il faut alors la décomposer.

**Exemple.** Quel type choisir pour la date de naissance d'une personne ?

On pourrait la représenter dans une chaîne (par ex : "17/3/1985") mais cela rendrait difficile le traitement, les calculs (par exemple, déterminer le numéro du mois). Le mieux est sans doute de la décomposer en trois parties : le jour, le mois et l'année, tous des entiers.

Plus tard, vous verrez qu'il est possible de définir de nouveaux types de données grâce aux *classes* en Java.

Il sera alors possible de définir et utiliser un type `Date` et il ne sera plus nécessaire de décomposer une date en trois morceaux bien que le type sera bien entendu composé de trois valeurs.

**3.3.4 Exercice**

Quel(s) type(s) de données utiliseriez-vous pour représenter :

- ▷ le prix d'un produit en grande surface ;
- ▷ la taille de l'écran de votre ordinateur ;
- ▷ votre nom ;
- ▷ votre adresse ;
- ▷ le pourcentage de remise proposé pour un produit ;
- ▷ une date du calendrier ;
- ▷ un moment dans la journée ?

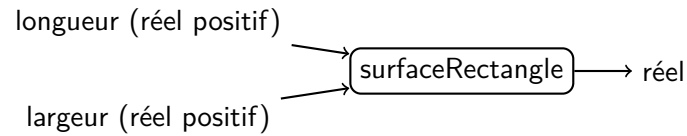
**3.4 Résumés**

Toutes les informations déjà collectées sur le problème peuvent être résumées.

**3.4.1 Résumé graphique**

Représentation graphique du problème.

**Exemple.** Pour le problème, de la surface du rectangle, on fera le schéma suivant :



### 3.4.2 Résumé textuel

Résumé textuel du problème, indiquant clairement les données et les résultats recherchés.

**Exemple.**

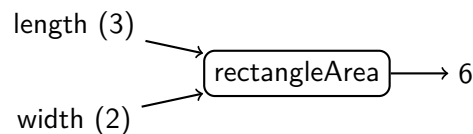
<b>Données</b>	longueur (un réel positif) largeur (un réel positif)
<b>Résultat</b>	la surface du rectangle

## 3.5 Exemples numériques

Une dernière étape pour vérifier que le problème est bien compris est de donner quelques exemples numériques. Il est possible de les spécifier en français, via un graphique ou via une notation compacte.

**Exemples.** Voici différentes façons de présenter des exemples numériques pour le problème de calcul de la surface d'un rectangle :

- ▷ En français : si la longueur du rectangle vaut 3 et sa largeur vaut 2, alors sa surface vaut 6.
- ▷ Via un schéma :



- ▷ En notation compacte : `rectangleArea(3, 2)` donne/vaut 6.

# Chapitre 4

## Premiers algorithmes

Dans le chapitre précédent, vous avez appris à analyser un problème et à clairement le spécifier. Il est temps d'écrire des solutions. Pour cela, nous allons devoir trouver comment passer des données au résultat et l'exprimer dans un langage compris de tous.

En fonction de la difficulté du problème et de notre sensibilité, nous pouvons représenter un algorithme de plusieurs manières ; en langage naturel (en français ou en anglais), en pseudocode ou avec un organigramme. . . Dans ce chapitre, nous présenterons les algorithmes en langage naturel, en pseudocode, avec un organigramme et en langage Java.

Dans la suite du cours, nous nous contenterons de les écrire en langage Java.

## Contenu

---

4.1	Exercice résolu : un problème simple . . . . .	<b>31</b>
4.1.1	Trouver l'algorithme . . . . .	31
4.1.2	Vérifier l'algorithme . . . . .	32
4.1.3	Écrire l'algorithme . . . . .	32
4.1.4	Remarques . . . . .	34
4.2	Résolution d'un second problème simple . . . . .	<b>35</b>
4.3	Décomposer les calculs ; variables et assignation . . . . .	<b>36</b>
4.3.1	Les variables . . . . .	36
4.3.2	L'assignation . . . . .	38
4.3.3	Tracer un algorithme . . . . .	39
4.3.4	Exercice résolu : durée du trajet . . . . .	40
4.4	Quelques difficultés liées au calcul . . . . .	<b>40</b>
4.4.1	Un peu de vocabulaire . . . . .	40
4.4.2	Les comparaisons et les assignations de variables booléennes . . . . .	41
4.4.3	Les opérations logiques . . . . .	42
4.4.4	La division entière et le reste . . . . .	45
4.4.5	Le hasard et les nombres aléatoires . . . . .	46
4.5	Des algorithmes et des programmes de qualité . . . . .	<b>47</b>
4.5.1	L'efficacité . . . . .	48
4.5.2	La lisibilité . . . . .	48
4.5.3	La rapidité . . . . .	48
4.5.4	La taille . . . . .	49
4.5.5	Conclusion . . . . .	49
4.6	Améliorer la lisibilité . . . . .	<b>49</b>
4.6.1	Mise en page des algorithmes et des programmes . .	49
4.6.2	Choix des noms . . . . .	51
4.6.3	Les commentaires . . . . .	51
4.6.4	Constantes . . . . .	52
4.7	Appel d'algorithme, appel de méthode . . . . .	<b>53</b>
4.8	Interagir avec l'utilisateur . . . . .	<b>54</b>
4.8.1	Afficher un résultat . . . . .	54
4.8.2	Demander des valeurs . . . . .	54
4.8.3	Préférer les paramètres . . . . .	55

---

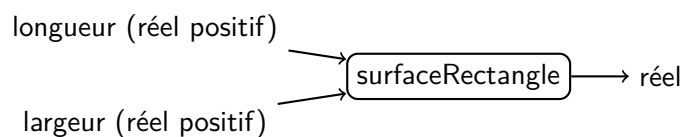
## 4.1 Exercice résolu : un problème simple

### 4.1.1 Trouver l'algorithme

Illustrons notre propos sur l'exemple qui a servi de fil conducteur tout au long du chapitre précédent. Rappelons l'énoncé et l'analyse qui en a été faite.

**Problème.** Calculer la surface d'un rectangle à partir de sa longueur et sa largeur.

**Analyse.** Nous sommes arrivés à la spécification suivante :



ou encore :

Données	longueur (un réel positif) largeur (un réel positif)
Résultat	la surface du rectangle

**Exemples.**

La surface du rectangle 3,2 vaut 6. La surface du rectangle 3.5, 1 vaut 3.5.

ou

▷ rectangleArea(3,2) donne 6 ;                      ▷ rectangleArea(3.5,1) donne 3.5.

ou

Données		
L	3	3.5
l	2	1
Résultat	6	3.5

Les exemples sont essentiels. Ce sont eux qui vont nous permettre de tester notre algorithme. Notre programme.

**Comment résoudre ce problème ?** La toute première étape est de comprendre le lien entre les données et le résultat. Ici, le lien est la formule du calcul d'une surface.

$$\text{surface} = \text{longueur} * \text{largeur}$$

La surface s'obtient donc en multipliant la longueur par la largeur <sup>22</sup>.

### 4.1.2 Vérifier l'algorithme

Une étape importante, après l'écriture d'un algorithme est la vérification de sa validité. Il est important d'exécuter l'algorithme avec des exemples numériques et vérifier que chaque réponse fournie est correcte.

Pour tester un algorithme il faut — même si ça paraît étrange — **éteindre son cerveau**. Il faut agir comme une machine et exécuter **ce qui est écrit** pas ce que l'on voulait écrire ou ce que l'on pensait avoir écrit ou encore ce qu'il est censé faire. Cela demande un peu de pratique.

**Exemple.** Vérifions notre solution pour le calcul de la surface du rectangle en reprenant les exemples choisis.

test n°	longueur	largeur	réponse attendue	réponse fournie	
1	3	2	6	6	✓
2	3.5	1	3.5	3.5	✓

### 4.1.3 Écrire l'algorithme

Pour décrire un algorithme, nous proposons plusieurs façons de faire, ayant l'objectif commun de pouvoir décrire cet algorithme à l'ordinateur dans un langage de programmation (Java). Il est important de pouvoir comprendre ces façons de faire (langage naturel, organigramme, pseudo-code) mais il n'y a pas d'obligation de toutes les utiliser.

#### 4.1.3.1 Langage naturel

En **langage naturel**, une solution aurait simplement cette allure :

La surface s'obtient grâce à la formule :

$$surface = longueur * largeur$$

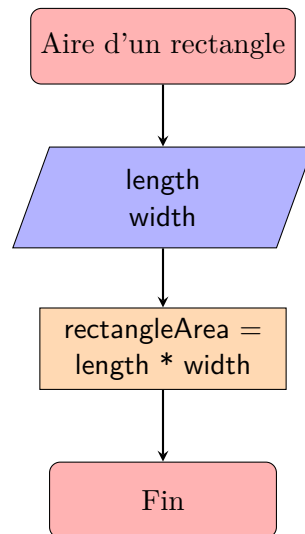
*langage naturel*

#### 4.1.3.2 Organigramme

Un **organigramme** d'une solution aura cette allure :

<sup>22</sup>. Trouver la bonne formule n'est pas toujours facile. Dans votre vie professionnelle, vous devrez parfois écrire un algorithme pour un domaine que vous connaissez peu, voire pas du tout. Il vous faudra alors chercher de l'aide, demander à des experts du domaine. Dans ce cours, nous nous concentrons sur des problèmes simples.



ORGANIGRAMME 2 – Solution de `rectangleArea`

#### 4.1.3.3 Pseudocode

Un **pseudocode** d'une solution pourrait s'écrire :

```

algorithm rectangleArea(length, width : reals) → real
  return length * width
  
```

*pseudocode*

Le mot **algorithm** et l'**indentation** — soulignée par une ligne verticale — permettent de délimiter l'algorithme. La première ligne est appelée l'**entête** de l'algorithme. On y retrouve :

- ▷ le nom de l'algorithme,
- ▷ une déclaration des données, qu'on appellera ici les **paramètres**,
- ▷ le type du résultat.

Les paramètres recevront des valeurs concrètes au **début** de l'exécution de l'algorithme.

L'instruction **return** permet d'indiquer la valeur du résultat, ce que l'algorithme *retourne*. Si on spécifie une formule, un calcul, c'est le **résultat** (on dit l'*évaluation*) de ce calcul qui est retourné et **pas la formule**.

Pour indiquer le calcul à faire, écrivez-le, naturellement comme vous le feriez en mathématique.

Pour demander l'**exécution** d'un algorithme (on dit aussi *appeler*) il suffit d'indiquer son nom et les valeurs concrètes à donner aux paramètres. Ainsi, `rectangleArea(6,3)` fait appel à l'algorithme correspondant pour calculer la surface d'un rectangle dont la longueur est 6 et la largeur est 3.

Dans ces notes, nous utiliserons «  $\rightarrow$  » pour montrer ce que retourne l'algorithme mais « : » ou un simple «  »<sup>23</sup> font l'affaire.

23. Ce symbole signifie : « ici se trouve une espace »

#### 4.1.3.4 Java

Une solution en Java s'écrit :

```
public class RectangleArea{  
    public static double rectangleArea(double length, double width){  
3        return length * width;  
    }  
}  
6
```

java

Telle quelle la solution n'est pas fonctionnelle en ce sens que l'exécution du programme ne montrera aucun résultat à l'écran.

Le **point d'entrée** d'un programme en Java est la méthode **main**. Cette méthode est la première qui sera exécutée. Elle est obligatoire si l'on veut pouvoir exécuter le programme.

Au minimum, si l'on veut calculer et afficher l'aire d'un rectangle de longueur 3 et de largeur 2, il faudrait plutôt écrire :

```
public class RectangleArea{  
    public static double rectangleArea(double length, double width){  
3        return length * width;  
    }  
  
6    public static void main(String[] args){  
        System.out.println(rectangleArea(3,2));  
    }  
9 }
```

java

#### 4.1.4 Remarques



**Attention :**

Écrire une solution complète d'un exercice c'est :

- ▷ spécifier le problème ;
- ▷ fournir des exemples, les tests ;
- ▷ construire son algorithme ;
- ▷ tester le programme et constater qu'il fournit bien les résultats attendus

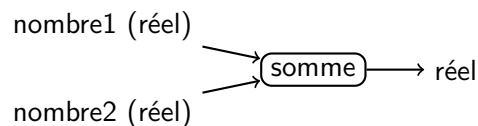
## 4.2 Résolution d'un second problème simple

Vous pouvez vous baser sur la fiche 1 page 230 qui résume la résolution du calcul de la surface d'un rectangle, depuis l'analyse de l'énoncé jusqu'à l'algorithme et à sa vérification.



**Problème.** Calculer la somme de deux nombres donnés.

**Analyse.**



ou :

Données	nombre 1 (un nombre réel quelconque) nombre 2 (un autre nombre réel)
Résultat	la somme des deux nombres

**Exemples** Les exemples qui deviendront les tests :

test n°	nombre1	nombre2	réponse attendue	réponse fournie
1	3	2	5	5
2	-3	2	-1	-1
3	3	2.5	5.5	5.5
4	-2.5	2.5	0	0

**Algorithme** L'algorithme s'écrit simplement en langage naturel :

somme = n1 + n2

*langage naturel*

La traduction en un programme Java est très semblable au premier exemple.

```

public class Addition{
2  public static double add(double number1, double number2){
    return number1 + number2;
  }
5 }
  
```

java

**Tester le programme** Il est aussi assez facile de vérifier qu'il fournit bien les bonnes réponses pour les exemples choisis.

test n°	nombre1	nombre2	réponse attendue	réponse fournie	
1	3	2	5	5	✓
2	-3	2	-1	-1	✓
3	3	2.5	5.5	5.5	✓
4	-2.5	2.5	0	0	✓

### 4.3 Décomposer les calculs ; variables et assignation

Pour pouvoir décomposer un calcul un peu long en plusieurs étapes nous devons introduire deux nouvelles notions : les *variables locales* et *l'assignation*.

Par exemple le calcul d'un prix TTC (toutes taxes comprises) d'un ensemble de produits dont on connaît la quantité et le prix unitaire hors taxe, peut se décomposer en le calcul du prix unitaire TTC et, ensuite, en le calcul du prix total.

#### 4.3.1 Les variables



Une **variable** est une zone mémoire munie d'un nom et qui contiendra une valeur d'un type donné. Cette valeur *peut* évoluer au fil de l'avancement de l'algorithme ou du programme. Elle sert à retenir des étapes intermédiaires de calculs.

Les variables peuvent être **locales** ou **globales**.

- ▷ Une **variable locale** n'est connue et utilisable qu'au sein de l'algorithme où elle est déclarée.

En Java, une variable locale n'est connue que dans le *bloc* d'instructions dans lequel elle est déclarée. C'est sa portée, son *scope*. Un bloc d'instructions est un ensemble d'instructions délimitées par une paire d'accolades.

- ▷ Une **variable globale** est connue dans tous les algorithmes d'un même problème.

En Java, une variable globale est accessible par toutes les classes et les méthodes d'un même projet. Nous y reviendrons en DEV2.

Dans nos algorithmes et dans nos programmes nos variables seront toujours locales.

Pour être utilisable, une variable doit être *déclarée* au début de l'algorithme. La **déclaration** d'une variable est l'instruction qui définit son nom et son type. On pourrait écrire :

Longueur et largeur seront les noms de deux objets destinés à recevoir  
Les longueur et largeur du rectangle, c'est-à-dire des nombres à valeurs  
réelles.

*langage naturel*

Nous pouvons abrégé – et passer à l’anglais – en :

```
reals length, width
```

*langage naturel*

Certains langages de programmation imposent que les variables soient déclarées — c’est le cas de Java — et le vérifient tandis que d’autres permettent d’utiliser des variables sans les déclarer.

Dans ce premier cours de développement, nous déclarerons toujours nos variables avant de les utiliser. Principalement dans un souci de lisibilité et également car ceci évite des erreurs lors de l’élaboration des algorithmes et des programmes.

En langage Java, cela donne :

```
double length, width;
```

java

Par convention, Java privilégie une déclaration par ligne, comme ceci :

```
1 double length;  
double width;
```

java

Pour choisir le nom d’une variable, les règles sont les mêmes que pour les données d’un problème.

### 4.3.2 L'assignation



L'**assignation** (on dit aussi *affectation interne*) est une instruction qui donne une valeur à une variable ou la modifie.

Cette instruction est probablement la plus importante car c'est ce qui permet de retenir les résultats de calculs intermédiaires.

Pour cette assignation, nous pourrions utiliser une flèche  $\leftarrow$  ou le symbole d'égalité  $=$ . Dans ces notes nous utiliserons le symbole d'égalité  $=$ .

```
variableName = expression
```

*langage naturel*

En Java, nous utiliserons exactement le même symbole  $=$  :

```
variableName = <une expression>
```

java

Bien que nous utilisions le symbole de l'égalité mathématique, l'assignation n'a pas le même sens.

- ▷ l'égalité mathématique définit que deux choses sont égales et le resteront ;
- ▷ l'assignation assigne une valeur – le membre de droite – à la variable se trouvant à gauche à un moment  $t$ .



Une **expression** est un calcul faisant intervenir des variables, des valeurs explicites et des opérateurs (comme  $+$ ,  $-$ ,  $<$ , ...). Une expression a une **valeur**.

**Exemples.** Quelques assignations correctes en algorithmique :

```
denRes = den1 * den2  
count = count + 1  
average = (number1 + number2) / 2  
isALowerthanB = a < b  
aString = "hello"
```

*langage naturel*

Et ces même exemples — corrects — en Java :

```
1 denRes = den1 * den2;  
  count = count + 1;  
  average = (number1 + number2) / 2;  
4 isALowerthanB = a < b;  
  aString = "hello";
```

java

**Remarques**

- ▷ Une assignation n'est ni une égalité, ni une définition.

Ainsi, l'assignation `cpt = cpt + 1` ne veut pas dire que `cpt` et `cpt + 1` sont égaux, ce qui est mathématiquement faux mais que la *nouvelle* valeur de `cpt` doit être calculée en ajoutant 1 à sa valeur actuelle.

Ce calcul doit être effectué au moment de l'exécution de cette instruction.

- ▷ Seules les variables déclarées peuvent être affectées.
- ▷ Toutes les variables apparaissant dans une expression doivent avoir été affectées préalablement. Le contraire provoquerait une erreur, un arrêt de l'algorithme ou du programme.
- ▷ La valeur affectée à une variable doit être compatible avec son type. Pas question de mettre une chaîne dans une variable booléenne.
- ▷ Certaines personnes utilisent  $\leftarrow$  comme symbole d'assignation pour les algorithmes. C'est très bien aussi.

Nous utiliserons dans ce cours le symbole `=` afin d'être plus proche du langage Java.

**4.3.3 Tracer un algorithme**

Pour vérifier qu'un algorithme est correct, le développeur ou la développeuse sera souvent amenée<sup>24</sup> à le tracer.

**Tracer** un algorithme ou un programme consiste à suivre l'évolution des variables à chaque étape ou à chaque instruction et à vérifier qu'elles contiennent bien à tout moment la valeur attendue.

Dans le cadre de nos algorithmes, ce traçage se fait sur papier comme dans les exemples ci-dessous. Par contre dans la panoplie des outils de développement liés à un langage de programmation apparaît toujours un débogueur. Un débogueur est un outil aidant le ou la développeur·se à trouver ses erreurs. La première manière de faire étant de tracer son code. Il est donc tout à fait possible d'exécuter un programme instruction par instruction et de voir le contenu des variables à chaque étape. Nous ne pouvons que vous encourager à le faire.

**Exemple.** Traçons des extraits d'algorithmes.

#	a	b	c
1	indéfini	indéfini	indéfini
2	12		
3		5	
4			7
5	19		
6		19	

```

int a, b et c;
a = 12;
3 b = 5;
c = a - b;
a = a + c;
6 b = a;

```

java

24. Je m'autorise un accord de proximité.

```

int a, b, c;
2 a = 12;
  c = a - b;
  d = c - 2;
5

```

java

#	a	b	c
1	indéfini	indéfini	indéfini
2	12		
3			???
4			???

c ne peut pas être calculé car b n'a pas été initialisé ; quant à d, il n'est même pas déclaré !

#### 4.3.4 Exercice résolu : durée du trajet

Savoir, face à un cas concret, s'il est préférable de décomposer le calcul ou pas, n'est pas toujours évident. La section 4.6 page 49 sur la lisibilité vous apportera des arguments qui permettront de trancher.

L'exercice suivant est suffisamment complexe pour mériter une décomposition du calcul.

**Exercice : Durée du trajet** Étant donné la vitesse moyenne non nulle en **m/s** d'un véhicule et la distance parcourue en **km** par ce véhicule, calculer la durée en secondes du trajet de ce véhicule.

Nous vous proposons de rédiger une solution complète.

Vous pouvez vérifier ensuite sur la fiche 2 page 232 qui présente une solution.

### 4.4 Quelques difficultés liées au calcul

Vous êtes habitués à effectuer des calculs. L'expérience nous montre toutefois que certains calculs posent des difficultés. Soit parce que ce sont des opérations peu utilisées, soit parce qu'elles ne sont habituellement pas sous la forme de calculs. Citons :

- ▷ assigner des valeurs booléennes en fonction de comparaisons ;
- ▷ manipuler les opérateurs logiques ;
- ▷ utiliser la division entière et le reste.

Parfois, le problème se situe au niveau de la compréhension du vocabulaire. Examinons ces situations une à une en fournissant des exemples et des exercices pour que cela devienne naturel.

#### 4.4.1 Un peu de vocabulaire

Quelques notions : Qu'est-ce qu'un opérateur ? Un opérande ? Rappelons et fixons ces notions.





**expression**

Une expression indique un calcul à effectuer (par exemple :  $(a + b) * c$ ). Une fois le calcul effectué (on dit qu'on *évalue* l'expression), on obtient une valeur, d'un certain type. Une expression est composée d'opérandes et d'opérateurs, elle a une valeur et un type.

**opérateur**

Un opérateur est ce qui désigne une opération. Exemple :  $+$  désigne l'addition.

**opérande**

Un opérande est ce sur quoi porte l'opération. Exemple : dans l'expression  $a+b$ ,  $a$  et  $b$  sont les opérandes. Un opérande peut être une sous-expression. Exemple : dans l'expression  $(a+b) * c$ ,  $(a+b)$  est l'opérande de gauche de l'opérateur  $*$  et  $c$  l'opérande de droite.

**unaire, binaire, ternaire**

Un opérateur qui agit sur deux opérandes (le plus fréquent) est qualifié de binaire. On rencontre aussi des opérateurs unaires (ex : le  $-$  dans l'expression  $-a$ ). En Java, vous rencontrerez aussi un opérateur ternaire (3 opérandes) mais ils sont plus rares.

**littéral**

Un littéral est une valeur notée explicitement (comme 12, 34.4, "bonjour")

**priorité**

Les opérateurs sont classés par priorité. Cela permet de savoir dans quel ordre les exécuter. Par exemple, la multiplication est prioritaire par rapport à l'addition. C'est pourquoi l'expression  $a + b * c$  est équivalente à  $a + (b * c)$  et pas à  $(a + b) * c$ . Les parenthèses permettent de modifier ou de souligner la priorité.

**Exercice : analyse d'expression** Voici une série d'expressions. Nous vous proposons d'identifier tous les opérateurs et leurs opérandes, d'indiquer si les opérateurs sont unaires ou binaires et d'identifier les littéraux.

Nous vous proposons aussi de fournir une version de l'expression avec le moins de parenthèses possibles et une autre avec un maximum de parenthèses (tout en respectant le sens de l'expression bien sûr et sans mettre de parenthèses redondantes).

- ▷  $a+1$
- ▷  $(a+b)*12-4*(-a-b)$
- ▷  $a+(b*12)-4*-a$

#### 4.4.2 Les comparaisons et les assignations de variables booléennes

$3 + 1$  est un calcul dont le résultat est 4, un entier. C'est sans doute évident.

$1 < 3$  est aussi un calcul dont le résultat est un *booléen*, vrai en l'occurrence. Ce résultat peut être assigné à une variable booléenne.

**Exemples.** Voici quelques assignations correctes :

```

1 boolean positive,
  adult,
  successful,
4 perfect;
positive = nb > 0;
adult = age >= 21;
7 successful = code >= 10;
perfect = errors == 0;

```

java

**Remarque** Pour tester l'égalité, nous utilisons le symbole `==` afin de le différencier du symbole `=` qui est le symbole de l'assignation.

**Exercices : écrire des expressions booléennes** Pour chacune des phrases suivantes, écrivez l'assignation qui lui correspond.

- ▷ La variable booléenne *negatif* doit indiquer si le nombre *montant* est négatif.
- ▷ Un groupe est complet s'il contient exactement 20 personnes.
- ▷ Un algorithme est considéré comme long si le nombre de lignes dépasse 20.
- ▷ Un étudiant a *la plus grande distinction* si sa cote est de 18/20 ou plus.

### 4.4.3 Les opérations logiques

Les opérateurs logiques agissent sur des expressions booléennes (variables ou expressions à valeurs booléennes) pour donner un résultat du même type.

opérateur	nom	description
NON	négation	vrai devient faux et inversement
AND	conjonction logique	vrai si les 2 conditions sont vraies
OR	disjonction logique	vrai si au moins une des 2 conditions est vraie

**Remarque** Nous utilisons AND et OR mais nous acceptons et comprenons ET et OU.

Ce tableau se résume en *tables de vérité* comme suit :

a	b	a AND b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a OR b
true	true	true
true	false	true
false	true	true
false	false	false

a	NON a
true	false
false	true

Ces opérateurs peuvent intervenir dans des expressions booléennes.

### Exemples

- ▷ tarifPlein =  $18 \leq \text{âge} \text{ ET } \text{âge} < 60$
- ▷ distinction =  $14 \leq \text{cote} \text{ ET } \text{cote} < 16$
- ▷ nbA3chiffres =  $100 \leq \text{nb} \text{ ET } \text{nb} \leq 999$
- ▷ tarifRéduit = NON tarifPlein
- ▷ tarifRéduit = NON ( $18 \leq \text{âge} \text{ ET } \text{âge} < 60$ )
- ▷ tarifRéduit =  $\text{âge} < 18 \text{ OU } 60 \leq \text{âge}$

En Java, le AND (ET) s'écrit `&&`<sup>25</sup> et le OR (OU), `||`<sup>26</sup>.

Écrire des calculs utilisant ces opérateurs n'est pas facile car le français nous induit souvent en erreur en nous poussant à utiliser un ET pour un OU et inversement ou bien à utiliser des raccourcis d'écriture ambigus<sup>27</sup>.

Par exemple, ne pas écrire : `tarifRéduit = âge < 18 OU ≥ 60`

**Loi de De Morgan.** Cette loi s'énonce :

« La négation d'une conjonction (AND) de deux propositions est la disjonction (OR) des deux négations. De même, la négation d'une disjonction de deux propositions est la conjonction des deux négations. »

... et s'écrit plus simplement :

$$\text{NON } (a \text{ AND } b) \Leftrightarrow \text{NON } a \text{ OR NON } b$$

$$\text{NON } (a \text{ OR } b) \Leftrightarrow \text{NON } a \text{ AND NON } b$$

Par exemple, ces trois propositions sont équivalentes :

$$\text{tarifRéduit} = \text{NON } (18 \leq \text{âge} \text{ AND } \text{âge} < 60)$$

$$\text{tarifRéduit} = (\text{NON } 18 \leq \text{âge}) \text{ OR } (\text{NON } \text{âge} < 60)$$

$$\text{tarifRéduit} = \text{âge} < 18 \text{ OR } 60 \leq \text{âge}$$

**Priorités et parenthèses.** L'opérateur NON est prioritaire sur les opérateurs AND et OR.

L'opérateur AND est prioritaire sur le OR.

Ainsi l'expression : `NON a OR b AND c` doit se comprendre : `(NON a) OR (b AND c)`. Il est toujours possible d'ajouter des parenthèses pour aider à la compréhension.

25. `&` est le caractère *ampersand* (esperluette en français)

26. `|` est le caractère *pipe* (barre verticale en français)

27. Vous noterez que le nombre de "et" et de "ou" dans cette phrase ne facilite pas sa compréhension ;)

**Évaluation paresseuse** L'évaluation paresseuse (*lazy evaluation*) ou évaluation court-circuitée est une évaluation qui s'arrête dès que le résultat est connu.

Les opérateurs AND et OR sont des opérateurs court-circuités. En particulier, si la première partie d'un AND est fausse, il n'est pas nécessaire de regarder le deuxième opérande; le résultat sera faux. De même si la première partie d'un OR est vraie; le résultat sera vrai quelle que soit la valeur de la deuxième partie.

Cette manière de fonctionner permet de gagner du temps dans l'évaluation et permet aussi d'éviter des erreurs. Ceci est vrai dans les algorithmes et — surtout — dans beaucoup de langages de programmation. Java utilise cette évaluation paresseuse.

### Exemples.

```
boolean ok;

3 // Génère une erreur si b est nul
  ok = 1/b < 0.1;

6 // Si b est nul, provoque une erreur et un arrêt
  ok = 1/b < 0.1 && b != 0;

9 // YES ! C'est dans le bon ordre si b est nul,
  // la condition est fausse et
  // ok reçoit false
12 ok = b != 0 && 1/b < 0.1;
```

java

Cette propriété sera abondamment utilisée dans le parcours de tableaux par exemple.

**Exercice : simplifier des expressions booléennes** Voici quelques assignations correctes du point de vue de la syntaxe mais contenant des lourdeurs d'écriture. Trouvez des expressions plus simples qui auront un effet équivalent.

- ▷ beautifulBoolean = adulte == vrai
- ▷ beautifulBoolean = adulte == faux
- ▷ beautifulBoolean = etudiant == vrai AND jeune == faux
- ▷ beautifulBoolean = NON (adulte == vrai) AND NON (adulte == faux)
- ▷ nbA3chiffres = NON (nb<100 OR nb≥1000)

**Exercice : expressions logiques** Pour chacune des phrases suivantes, écrivez l'assignation qui lui correspond.

- ▷ J'irai au cinéma si le film me plait et que j'ai 20€ en poche.
- ▷ Je n'irai pas au cinéma si je n'ai pas 20€ en poche.
- ▷ Je brosserai le premier cours de la journée s'il commence à 8h et aussi si je n'ai pas dormi mes 8h.

#### 4.4.4 La division entière et le reste

La **division entière** consiste à effectuer une division en ne gardant que la partie entière du résultat. Le **reste** de la division entière de  $a$  par  $b$  est ce qui n'a pas été repris dans la division... ce qu'il reste.



Lorsque l'on fait une division entière, le **dividende**  $D$  est divisé par le **diviseur**  $d$  pour donner un **quotient**  $q$  et, éventuellement, un **reste**  $r$ .

$$D = d * q + r$$

$$\frac{D}{d} = q$$

et, éventuellement un reste  $r$

La division entière est souvent notée DIV et le reste MOD.

- ▷  $a \text{ DIV } b$  est le quotient de la division et
- ▷  $a \text{ MOD } b$  — nous dirons *a modulo b* — est le reste de cette division.

En Java, la division se note  $/$  et le reste  $\%$ .

Par exemple, imaginons qu'une classe comprenne 14 étudiants et étudiantes qu'il faut réunir par 3 dans le cadre d'un travail de groupe. Il est possible de former 4 groupes mais il restera 2 étudiants et étudiantes ne pouvant former un groupe complet. C'est le reste de la division de 14 par 3.

**Exemples :**

- |                  |                  |
|------------------|------------------|
| ▷ 7 DIV 2 vaut 3 | ▷ 7 MOD 2 vaut 1 |
| ▷ 8 DIV 2 vaut 4 | ▷ 8 MOD 2 vaut 0 |
| ▷ 6 DIV 6 vaut 1 | ▷ 6 MOD 6 vaut 0 |
| ▷ 6 DIV 7 vaut 0 | ▷ 6 MOD 7 vaut 6 |

##### 4.4.4.1 Utilité - Tester la divisibilité

Les deux opérateurs MOD et DIV, respectivement  $/$  et  $\%$ , permettent, par exemple, de tester si un nombre est un multiple d'un autre.

Si l'on veut savoir si un nombre est **pair** — pour rappel, un nombre pair est un multiple de 2 — il suffit de vérifier que le reste de la division par 2 est nul.

$$\text{nb pair} \quad \equiv \quad \text{nb divisible par 2} \quad \equiv \quad \text{nb MOD 2} = 0$$

En supposant que `isEven` (*even* pour pair et *odd* pour impair) est une variable booléenne, nous pourrions donc écrire : `isEven = nb MOD 2 == 0`.

Et, en langage java :

```
boolean isEven;
isEven = nb % 2 == 0;
```

java

#### 4.4.4.2 Utilité - Extraire les chiffres d'un nombre

Faisons une petite expérience numérique.

calcul	résultat	calcul	résultat
65536 MOD 10	6	65536 DIV 10	6553
65536 MOD 100	36	65536 DIV 100	655
65536 MOD 1000	536	65536 DIV 1000	65
65536 MOD 10000	5536	65536 DIV 10000	6

Nous voyons que les divisions entières (DIV) et les restes (MOD) avec des puissances de 10 permettent de conserver les chiffres de droite (division entière, MOD) ou d'enlever les chiffres de droite (DIV). Combinés, ils permettent d'extraire n'importe quel chiffre d'un nombre.

**Exemple :**  $(65536 \text{ DIV } 100) \text{ MOD } 10 = 5 \dots$  qui est le chiffre des centaines.

#### 4.4.5 Le hasard et les nombres aléatoires

Il existe de nombreuses applications qui font intervenir le hasard.

Par exemple dans les jeux où il est nécessaire de mélanger des cartes, lancer des dés, faire apparaître des ennemis de façon aléatoire...

Le vrai hasard n'existe pas en informatique puisqu'il s'agit de suivre des étapes précises dans un ordre fixé et que la machine est déterministe. Pourtant, on peut concevoir des algorithmes et des programmes qui *simulent* le hasard<sup>28</sup>. À partir d'un nombre donné<sup>29</sup> (appelé *graine* ou *seed* en anglais) ils fournissent une suite de nombres qui *ont l'air* aléatoires.

Concevoir de tels algorithmes est très compliqué et dépasse largement le cadre de ce cours. Heureusement, la plupart des langages informatiques proposent de base une façon d'obtenir un tel nombre aléatoire.

En Java, il suffit d'écrire<sup>30</sup>

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

java

pour obtenir un nombre (pseudo-)aléatoire compris entre 0 et 1 strictement. C'est-à-dire strictement plus petit que 1.

Dans nos algorithmes, nous pouvons toujours supposer qu'il existe un tel algorithme sans devoir l'écrire.

random → réel (entre 0 inclus et 1 exclu)

28. Pour être précis, nous devrions dire pseudo-hasard ou algorithmes et programmes pseudo-aléatoires.

29. Ce nombre peut être fixé ou généré à partir de l'environnement (par exemple, l'horloge interne).

30. Nous verrons plus tard qu'il existe une autre manière de faire en utilisant la classe `Random`

À partir de cet algorithme et de cette méthode découlent ces deux autres « algorithmes » :

- ▷ un entier entre 0 inclus et n exclu

(la partie entière de)  $\text{random()} * n$

*langage naturel*

- ▷ un entier entre min et max inclus

(la partie entière de)  $\text{min} + \text{random()} * (\text{max} - \text{min} + 1)$

*langage naturel*

En langage Java, il existe une méthode `Math.random()` comme nous l'avons dit mais également une autre manière de faire en utilisant la classe **Random** comme ceci<sup>31</sup> :

```
1 Random R = new Random();

// Une valeur entière comprise entre 0 et 13 strictement
4 int value = R.nextInt(13);

// Une valeur entière comprise entre 10 et 20 strictement
7 int value2 = 10 + R.nextInt(10);

// Une valeur pseudo-réelle entre 0 et 1 strictement
10 double value3 = R.nextDouble();
```

java

## 4.5 Des algorithmes et des programmes de qualité

Dans la section précédente, nous avons vu qu'il est possible de décomposer un calcul en étapes. Mais quand faut-il le faire ? Ou, pour poser la question autrement :

**Puisqu'il existe plusieurs algorithmes qui résolvent un problème, lequel préférer ? Quelle traduction dans un langage de programmation (Java) privilégier ?**

Répondre à cette question, c'est se demander ce qui fait la qualité d'un algorithme ou d'un programme informatique. Quels sont les critères qui permettent de juger ?

C'est un vaste sujet mais nous voudrions aborder les principaux.

La connaissance de l'algorithmique permet d'écrire des algorithmes de qualité. La connaissance d'un langage — qui est une autre compétence — permet d'écrire des programmes de qualité. Les qualités de l'un n'étant pas nécessairement celles de l'autre.

31. Le code n'est pas fonctionnel en l'état. Il manque un *import*. Nous y reviendrons

### 4.5.1 L'efficacité

L'**efficacité**<sup>32</sup> désigne le fait que l'algorithme (le programme) résout bien le problème donné. C'est un minimum !

### 4.5.2 La lisibilité

La **lisibilité** indique si une personne qui lit l'algorithme ou le programme peut facilement percevoir comment il fonctionne. C'est crucial car un algorithme ou un programme est **souvent lu** par de nombreuses personnes :

- ▷ celles qui doivent se convaincre de sa validité avant de passer à la programmation ;
- ▷ celles qui doivent trouver les causes d'une erreur lorsque celle-ci a été rencontrée<sup>33</sup> ;
- ▷ celles qui doivent faire évoluer l'algorithme ou le programme suite à une modification du problème ;
- ▷ et, accessoirement, celles qui doivent le coter ;)

C'est un critère **très important** qu'il ne faut surtout pas sous-évaluer. Vous en ferez d'ailleurs l'amère expérience : si vous négligez la lisibilité de votre algorithme ou de votre programme, vous-même ne le comprendrez plus quand vous le relirez quelque temps plus tard !

Comparer la lisibilité de deux algorithmes ou de deux programmes n'est pas une tâche évidente car c'est une notion subjective. Il faut se demander quelle version va être le plus facilement comprise par la majorité des lecteurs. La section 4.6 page suivante explique ce qui peut être fait pour rendre ses algorithmes plus lisibles. À ces recommandations s'ajouteront les **conventions d'écriture** propres à chaque langage qu'il faudra aussi respecter.

### 4.5.3 La rapidité

La **rapidité** indique si l'algorithme ou le programme permet d'arriver plus ou moins vite au résultat.

C'est un critère qui est souvent sur-évalué, essentiellement pour deux raisons.

- ▷ Il est trompeur. On peut croire une version plus rapide alors qu'il n'en est rien. Par exemple, on peut se dire que décomposer un calcul ralentit un programme puisqu'il doit gérer des variables intermédiaires. Ce n'est pas forcément le cas. Les compilateurs modernes sont capables de nombreuses prouesses pour optimiser le code et fournir un résultat aussi rapide qu'avec un calcul non décomposé.
- ▷ L'expérience montre que la recherche de rapidité mène souvent à des algorithmes moins lisibles. Or la lisibilité doit être privilégiée à la rapidité car sinon il sera impossible de corriger et/ou de faire évoluer l'algorithme.

Ce critère est un cas particulier de l'*efficacité* qui traite de la gestion économe des ressources. Vous reverrez l'idée de rapidité dans d'autres cours, avec la notion de *complexité* des algorithmes,

---

<sup>32</sup>. À ne pas confondre avec l'*efficacité* qui indique qu'il est économe en ressources.

<sup>33</sup>. On parle du processus de *déverminage* (ou *debugging* en anglais).



Le langage Java est réputé lent. C’est une affirmation péremptoire et peut-être périmée. Bien sûr, comme le *bytecode* est interprété (ces notions sont expliquées dans la section 5.1.1 p. 58) , il est nécessaire de charger la machine virtuelle. La gestion de la mémoire prise en charge par le langage a aussi un certain cout. Hormis ces deux aspects, la rapidité d’un programme dépend surtout de la qualité du code... et donc du développeur.

L’important, dans ce premier cours de développement est d’écrire des programmes lisibles et respectant les conventions d’écriture.

#### 4.5.4 La taille

Nous voyons parfois des étudiants et des étudiantes contentes d’avoir pu écrire un algorithme en moins de lignes. Ce critère n’a **aucune importance** ; un algorithme plus court n’est pas nécessairement plus rapide ni plus lisible.

Lors de la traduction en langage Java, certaines formes d’écriture — plus courtes — seront privilégiées. Nous y reviendrons et – au fur et à mesure de l’apprentissage – elles deviendront vite évidentes.

#### 4.5.5 Conclusion

Tous ces critères n’ont pas le même poids. Le point le plus important est bien sûr d’écrire des algorithmes et des programmes corrects mais ne vous arrêtez pas là ! Demandez-vous s’il n’est pas possible de le retravailler pour améliorer sa lisibilité<sup>34</sup>.

### 4.6 Améliorer la lisibilité

Comme nous venons de le voir, la lisibilité est une qualité essentielle que doivent avoir nos algorithmes et nos programmes. Qu’est ce qui permet d’améliorer la lisibilité d’un algorithme ?

#### 4.6.1 Mise en page des algorithmes et des programmes

Il y a d’abord la **mise en page** qui aide le lecteur à avoir une meilleure vue d’ensemble de l’algorithme, à en repérer rapidement la structure générale. Ainsi, dans ce syllabus :

- ▷ les mots imposés ou, tout au moins importants, sont mis en évidence (en gras<sup>35</sup>) ;
- ▷ une seule instruction se trouve par ligne ;
- ▷ les instructions à l’intérieur de l’algorithme sont *indentées* (décalées vers la droite). On indentera également les instructions à l’intérieur des choix et des boucles ;
- ▷ Le début et la fin de quelque chose — un bloc — sont marqués par des accolades () ou une ligne.

---

34. On appelle *refactorisation* l’opération qui consiste à modifier un algorithme ou un code sans changer ce qu’il fait dans le but, notamment, de le rendre plus lisible.

35. Difficile de mettre en gras avec un bic. Dans une version écrite vous pouvez : souligner ou surligner le mot, l’écrire en majuscule ou le mettre en couleur.

## Exemples

On n'écrira pas :



```
1 public static double travelTime(double speedMS, double distanceKm) {
  double distanceM;
  distance M = 1000 * distanceKM;
4 return distanceM / speedMS;
}
```

java

ni :



```
public static double travelTime(double speedMS, double distanceKm) {
  double distanceM;
3   distance M = 1000 * distanceKM; return distanceM / speedMS; }
```

java

mais plutôt :

```
public static double travelTime(double speedMS, double distanceKm) {
2  double distanceM;
  distance M = 1000 * distanceKM;
  return distanceM / speedMS;
5 }
```

java

Plus spécifiquement pour les programmes, l'utilisation d'un éditeur de code<sup>36</sup> ou d'un IDE (Environnement de Développement Intégré)<sup>37</sup> aide à l'écriture de programmes lisibles et respectant les conventions.

- ▷ Le code est indenté comme les algorithmes.
- ▷ Une seule instruction par ligne.
- ▷ La longueur des lignes n'excède pas 80 caractères<sup>38</sup>.
- ▷ S'il est nécessaire de couper une ligne trop longue, celle-ci est coupée avant un opérateur ou après une virgule.

36. Un éditeur de code est un programme aidant à l'édition de code qu'il ne faut pas confondre avec un éditeur de texte. Des éditeurs de codes connus ; *gVim*, *Notepad++*

37. Des IDE connus ; *Netbeans*, *Eclipse*...

38. Ce n'est pas la largeur de l'écran qui détermine la longueur des lignes. La limite à 80 caractères permet un code plus lisible. Elle permet aussi de pouvoir lire plus vite le code en limitant le mouvement des yeux de gauche à droite.

```

public static boolean iDoNothing(double beautifulDouble,
    int firstInteger, boolean isItReallyTrue){
3   double notSoBeautiful = beautifulDouble * firstInteger;
    return notSoBeautiful > 100 && notSoBeautiful < 10000
        || isItReallyTrue;
6 }

```

java

#### 4.6.2 Choix des noms

Il y a, ensuite, l'écriture des instructions elles-mêmes. Ainsi :

- ▷ Il faut choisir soigneusement les noms (d'algorithmes, de paramètres, de variables...)
- ▷ Il faut décomposer (ou au contraire fusionner) des calculs pour arriver au résultat jugé le plus lisible.
- ▷ Il est également possible d'introduire des commentaires et/ou des constantes. Deux concepts que nous allons développer maintenant.

#### 4.6.3 Les commentaires

**Commenter** un algorithme signifie lui ajouter du texte explicatif destiné au **lecteur** pour l'aider à mieux comprendre le fonctionnement de l'algorithme. Un commentaire n'est pas utilisé par celui qui exécute l'algorithme ; il ne modifie pas ce que l'algorithme fait.



Habituellement, on distingue deux sortes de commentaires :

- ▷ Ceux placés **au-dessus** de l'algorithme ou du programme qui expliquent **ce qu'il fait** et dans quelles **conditions** il fonctionne (les contraintes sur les paramètres).  
C'est la documentation **documentation**.
- ▷ Ceux placés **dans** l'algorithme ou le programme qui expliquent **comment** il le fait.

Commenter correctement un programme est une tâche qui n'est pas évidente et qu'il faut travailler. Il faut arriver à apporter au lecteur une information **utile** qui n'apparaît pas directement dans le code. Par exemple, il est contre-productif de répéter ce que l'instruction dit déjà. Il faut supposer que le lecteur connaît le langage et l'algorithmique.

Dans nos algorithmes, les commentaires commencent par `//`.

En langage Java, il y a trois manières de commenter :

1. `//` en début de ligne ;
2. le commentaire sur plusieurs lignes en commençant par `/*` et en le terminant par `*/` ;  
`/* <commentaire> */`
3. le commentaire sur plusieurs lignes en commençant par `/**` et en le terminant par `*/`. Dans ce cas, c'est un commentaire destiné à la documentation *javadoc*. Nous y reviendrons (dans la section 16 p.181).

Voici quelques mauvais commentaires



```
double length    // La longueur est un réel
2 /* La somme est initialisée à 0.
   Ce sera toujours le cas. */
int sum = 0;
5
```

java

**Remarque** Un excès de commentaires peut être le révélateur des problèmes de lisibilité du code lui-même. Par exemple, un choix judicieux de noms de variables peut s'avérer bien plus efficace que des commentaires. Ainsi, écrire :

```
newCapital = oldCapital * ( 1 + rate / 100 )
```

*langage naturel*

sans commentaires est bien préférable aux lignes suivantes :

```
c1 = c0 * ( 1 + r/100)
Calcul du nouveau capital en appliquant le taux
```



*langage naturel*

Pour résumer :

**N'hésitez pas à documenter votre programme pour expliquer ce qu'il fait et à le retravailler pour que tout commentaire à l'intérieur de l'algorithme devienne superflu.**

**Exemples.** Voici comment on pourrait documenter un de nos algorithmes.

```
1 /*
   * Calcule la surface d'un rectangle dont on donne la largeur et
   * la longueur.
4  *
   * Les données ne sont pas négatives.
   */
7 public static double rectangleArea (double length, double width){
   return length * width;
}
10
```

java

#### 4.6.4 Constantes

Une **constante** est une information pour laquelle nom, type et valeur sont figés.

L'usage est d'écrire les constantes en majuscules.

L'utilisation de constantes dans vos algorithmes présente les avantages suivants :

- ▷ une meilleure lisibilité du code, pour autant que vous lui trouviez un nom explicite ;

- ▷ une plus grande facilité pour modifier le code si la constante vient à changer (modification légale du seuil de réussite par exemple).

### Exemples

```
public static final double PI = 3.1415;
2 public static final int PASS_LEVEL = 10;
public static final String ESI = "École_supérieure_d'Informatique";
```

java

**Remarque** En Java, définir une constante  $\pi$  est un mauvais choix. Cette constante existe déjà. Elle est accessible directement par `Math.PI`.

**Exercice** Utiliser une constante. Trouvez un algorithme que vous avez écrit où l'utilisation de constante pourrait améliorer la lisibilité de votre solution.

## 4.7 Appel d'algorithme, appel de méthode

Reprenons l'algorithme `rectangleArea` qui nous a souvent servi d'exemple. Il permet de calculer la surface d'un rectangle dont on connaît la longueur et la largeur. Mais d'où viennent les données ? Et que faire du résultat ?

Tout d'abord, un algorithme ou un programme peut utiliser (on dit **appeler**) un autre algorithme ou programme.

Cet autre algorithme doit exister *quelque part* : sur la même page, une autre page, un autre document, peu importe.

En Java, la contrainte est un peu plus forte. Les programmes sont généralement écrits dans plusieurs fichiers qui doivent « se retrouver »... mais dans un premier temps, nous nous contenterons d'un seul. Ce fichier représentera une *classe*.

- ▷ Les instructions qui définissent une classe sont :

```
public class MyClass{
2  // statements
}
```

java

- ▷ La classe `MyClass` doit se trouver dans le fichier `MyClass.java`<sup>39</sup>

Pour pouvoir appeler une méthode en Java qui se trouve dans la même classe, il suffit d'écrire :

```
area = rectangleArea(122, 3.78);
2
```

java

<sup>39</sup>. Cette contrainte peut être relâchée à partir de JDK11. Voir <http://namok.be/blog/?post/2019/04/06/JDK10-11-12>

L'appel d'une méthode est considéré comme une expression, un calcul qui, comme toute expression, possède une valeur (la valeur retournée) et un type. Elle peut intervenir dans un calcul plus grand, être assignée à une variable. . .

## 4.8 Interagir avec l'utilisateur

### 4.8.1 Afficher un résultat

Un programme concret (en Java par exemple) qui permet de calculer des surfaces de rectangles devra communiquer le résultat à l'utilisateur du programme. Nous allons renseigner un affichage par la commande **print**. Ce qui donne :

```
print 2  
print rectangleArea(122, 3.78)  
  
afficher 3
```

*langage naturel*

En langage naturel, nous pouvons choisir d'écrire afficher, print, écrire à l'écran. . . Dans ces notes, nous écrirons **print**.

L'instruction **print** signifie que l'algorithme doit, à cet endroit communiquer une information à l'utilisateur. La façon dont il va communiquer cette information (à l'écran dans une application texte, via une application graphique, sur un cadran de calculatrice ou de montre, sur une feuille de papier imprimée, via un synthétiseur vocal. . .) ne nous intéresse pas ici.

En langage Java, un affichage sur la *sortie standard*, la console ou encore dans le terminal se fait comme suit :

```
1 System.out.println(rectangleArea(122, 3.78));
```

**java**

Les lettres *ln* qui suivent le mot *print* signifient *new line* et indiquent un passage à la ligne après l'affichage. Il existe aussi une instruction **print** sans les lettres *ln*. . . qui ne passe pas à la ligne.

Avec un organigramme, les affichages et les demandes peuvent se faire à l'aide d'un parallélogramme.

Afficher rectangleArea(122, 3.78)

### 4.8.2 Demander des valeurs

Il serait maintenant intéressant de demander à l'utilisateur ce que valent la longueur et la largeur. C'est le but de la commande **read**.

```
read length
read width
print rectangleArea(length, width)
```

demander la valeur de length

*langage naturel*

En langage naturel, nous pouvons choisir d'écrire lire, read, demander la valeur de... Dans ces notes, nous écrirons **read**.

L'instruction **read** signifie que l'utilisateur va, à cet endroit de l'algorithme, être sollicité pour donner une valeur qui sera affectée à une variable. À nouveau, la façon dont il va indiquer cette valeur (au clavier dans une application texte, via un champ de saisie ou une liste déroulante dans une application graphique, via une interface tactile, via des boutons physiques, via la reconnaissance vocale...) ne nous intéresse pas ici.

En langage Java, trois instructions seront nécessaires pour pouvoir faire une lecture au clavier pour un programme s'exécutant dans la console<sup>40</sup>.

```
1 import java.util.Scanner;
  // ...
  Scanner keyboard = new Scanner(System.in);
4 // ...
  double length = keyboard.nextDouble();
```

java

### 4.8.3 Préférer les paramètres

Un algorithme avec paramètres est toujours plus intéressant qu'un algorithme qui demande les données et affiche le résultat car il peut être utilisé (appelé) dans un autre algorithme pour résoudre une partie du problème. C'est exactement pareil pour un programme ; on privilégiera des méthodes recevant des valeurs en arguments que des méthodes qui demandent les données à l'utilisateur. Cette demande sera faite à un autre moment.

L'exemple du rectangle pourrait s'écrire de manière un peu plus complète comme suit :

---

40. Pour une application graphique, c'est encore un peu plus compliqué.

```
import java.util.Scanner;

3 public class AreaTest {
    /*
    * Calcule la surface d'un rectangle dont on donne la largeur et
6    * la longueur.
    *
    * Les données ne sont pas négatives.
9    */
    public static double rectangleArea(double length, double width){
        return length * width;
12    }

    public static void main(String[] args){
15        double length;
        double width;
        Scanner keyboard = new Scanner(System.in);

18        System.out.println("Entrez la longueur:");
        length = keyboard.nextDouble();
21        System.out.println("Entrez la largeur:");
        width = keyboard.nextDouble();

24        System.out.println("Surface: " + rectangleArea(length, width));
    }
27 }
```

java



# Chapitre 5

## Premiers programmes

La traduction d'un algorithme en un programme est une première étape de développement. La réalisation — c'est-à-dire l'exécution du programme sur une machine — est une seconde étape très importante. Que faut-il faire en plus de la traduction de l'algorithme pour que le programme fonctionne sur un ordinateur ?

### Contenu

5.1	Introduction . . . . .	<b>57</b>
5.1.1	Compilation - interprétation . . . . .	58
5.2	Environnement de développement . . . . .	<b>61</b>
5.2.1	La base . . . . .	61
5.2.2	Écrire le code . . . . .	61
5.2.3	Exécuter le programme . . . . .	62
5.2.4	Hello world . . . . .	62
5.3	La grammaire du langage . . . . .	<b>63</b>

### 5.1 Introduction

Il existe beaucoup de langages de programmation. Ces langages peuvent être rassemblés par classes de langages en fonction de leurs spécificités. Une classe de langages est adaptée à une classe de problèmes. Les langages — comme les problèmes — évoluent au fur et à mesure du temps.

#### langage machine

le langage machine est le langage de plus bas niveau. Il est exécutable par la machine et incompréhensible par l'humain sans effort ;

#### langage assembleur

là où tout est représenté par des nombres en langage machine, le langage assembleur propose une première couche d'abstraction : les instructions sont des mots : MOV, JMP...

#### langage de haut niveau

les langages de haut niveau sont destinés aux développeurs, le niveau d'abstraction est grand. Ils proposent des instructions, des variables, des structures de contrôle... tout ça sera traduit pour être compris par la machine.

Par exemple, *Fortran*, *COBOL*, *Pascal*, *C*...

### langage orienté objets

là où les langages de haut niveau étaient plus orientés sur les problèmes à résoudre

— Que faut-il faire ?

les langages orientés objets s'intéressent d'abord aux données

— Quelles sont les données que nous avons et que devons nous fournir ?

C'est sans doute la classe de langages la plus répandue.

Par exemple, *C++*, *Java*, *C#*, *Python*, *Go*, *Ruby*, *VB.NET*, *Vala*, *Objective C*, *Eiffel*, *Ada*, *PHP*, *Smalltalk*, *Scala*...

### langage fonctionnel

là où les langages orientés objets s'intéressent aux changements d'état des objets lorsqu'ils évoluent au fur et à mesure du programme, la programmation fonctionnelle consiste à exprimer le problème à résoudre en terme de fonctions (mathématiques). C'est une autre façon de programmer. Si elle est peu répandue, elle n'est pourtant pas récente.

Par exemple, *Lisp*, *Common Lisp*, *Haskell*, *Scala*...

Dans ce cours, nous nous intéressons au langage orienté objets, Java.

Le langage Java est un langage strict. Il impose, par exemple, que l'on déclare les variables que l'on utilise et il vérifie que les données assignées aux variables soient du « bon » type. Il possède un *garbage collector* (ramasse-miettes en français<sup>41</sup>) qui gère la mémoire à la place du développeur. Ces aspects facilitent l'acquisition de bonnes pratiques de développement.

C'est également un langage très présent dans l'industrie. Il est donc bien adapté pour des étudiants entamant un *bachelor* professionnalisant.

Certaines personnes lui reprocheront sa lenteur et sa syntaxe qui peut paraître lourde au premier abord tandis que d'autres rétorqueront que la lenteur n'est due qu'à la mauvaise qualité du code.

Vous aurez tout le loisir de vous faire votre propre idée.

#### 5.1.1 Compilation - interprétation

L'ordinateur ne comprenant que le langage machine, toutes les instructions devront, à un moment ou à un autre, être traduites en langage machine. Pour certains langages, la traduction se fait une fois pour toutes. Ils sont compilés. Pour d'autres la traduction se fait au *fil de l'eau*, pendant l'exécution. Ils sont interprétés.

---

41. Vous noterez la traduction.

**Définition**

Un langage est **compilé** (voir fig. 5.1) si le code source est traduit d'une traite. Un nouveau fichier contenant le code exécutable est créé.

Ces langages nécessitent un **compilateur**.

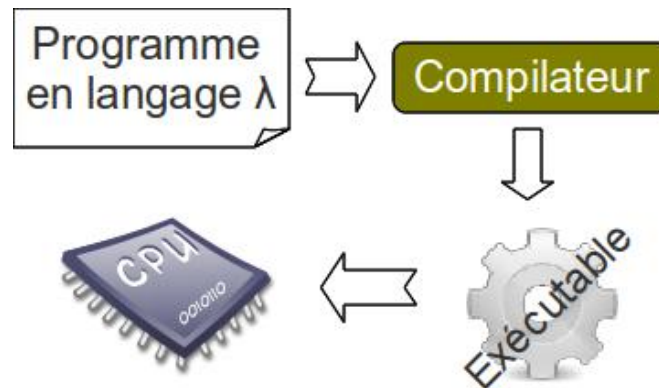


FIGURE 5.1 – Langage compilé

**Définition**

Un langage est **interprété** (voir fig. 5.2) si le code source est traduit instruction par instruction. Aucun nouveau fichier n'est créé et les instructions sont traduites à chaque exécution du programme.

Ces langages nécessitent un **interpréteur**<sup>42</sup>.

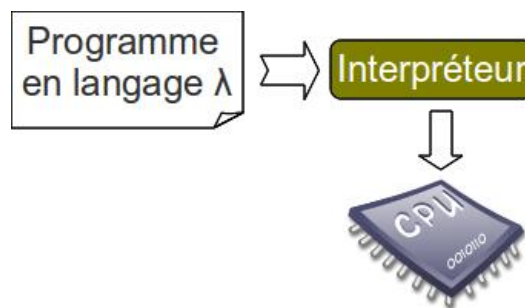


FIGURE 5.2 – Langage interprété

Les deux approches ont leurs avantages et leurs inconvénients.

- ▷ Dès lors qu'une machine possède l'interpréteur, le développeur peut diffuser son code et le code pourra être exécuté sur la machine... quel que soit son OS (*operating system*, système d'exploitation).  
Le programme fonctionnera sans modification supplémentaire sur une machine MS Windows, Linux ou Mac OS.
- ▷ Avec une approche « interprétée », pour diffuser un programme, il faut diffuser le code source. S'il s'agit d'un langage compilé, il est possible de ne distribuer que le binaire / exécutable.

42. Certaines personnes préfèrent parler d'un interprète mais le terme prête à confusion.

Par contre, il existe des « *décompilateurs* » qui peuvent reconstruire le code source à partir du binaire / exécutable.

- ▷ La phase de compilation permet de détecter beaucoup de (petites) erreurs avant d'exécuter le programme.
- ▷ L'exécution du binaire / exécutable est plus rapide que l'interprétation du code source.



Java a une approche mixte, il est compilé et ensuite interprété. Le code source est compilé et produit un *bytecode* sauvegardé dans un nouveau fichier. Le *bytecode* est ensuite interprété par une machine virtuelle, la *jvm* (*Java Virtual Machine*).

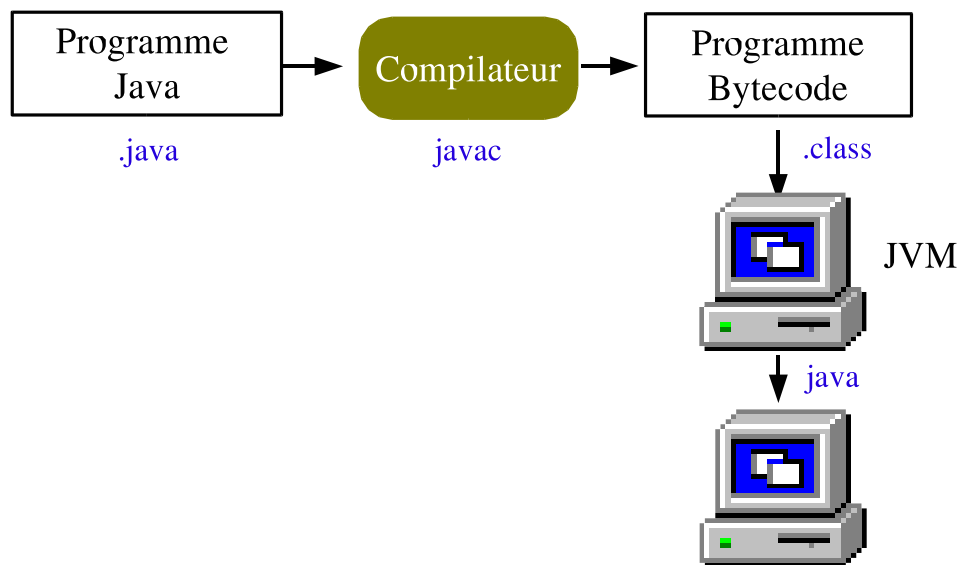


FIGURE 5.3 – Java est compilé et ensuite interprété

Le **code source** du programme est écrit dans un fichier texte dont l'extension sera `.java`. C'est le fichier que manipule le développeur. Le code source est la suite d'instructions java.

À l'aide du **compilateur** — le programme `javac` — le fichier est compilé. Un nouveau fichier ayant comme extension `.class`, est créé contenant le *bytecode*. Ce fichier est destiné à la machine virtuelle. Voici l'instruction pour compiler le programme.

```
$  
javac MyProgram.java
```

terminal

C'est la machine virtuelle — le programme `java` — qui est l'interpréteur du bytecode. C'est ce programme qui permet d'exécuter le... programme.

```
$  
java MyProgram
```

terminal

### Remarques

1. Notez que `javac` prend le nom d'un fichier en paramètre (avec son extension) alors que `java` prend le nom d'un programme — nous dirons une classe bientôt — en paramètre (sans extension donc).
2. Avec JDK11, il est possible de passer un fichier à la commande `java`. Dans ce cas, la commande compilera le fichier, conservera le bytecode en RAM et l'exécutera. Il n'y aura donc pas création d'un fichier contenant le *bytecode* et le code source sera exécutable en une seule commande (voir [\[Bet19\]](#)).

```
$  
java MyProgram.java
```

terminal

Attention de ne pas confondre.

## 5.2 Environnement de développement

L'environnement de développement représente l'ensemble des outils nécessaires au développement, à l'écriture des programmes.

### 5.2.1 La base

Il est évidemment nécessaire d'avoir un **ordinateur** et de le connaître un tant soit peu. Voici quelques compétences nécessaires :

- ▷ manipuler des fichiers : les déplacer, les trouver, les renommer...
- ▷ ouvrir un terminal ;
- ▷ connaître son clavier et particulièrement où se trouvent les caractères spéciaux. Par exemple : `{ } [ ] ; < > " ' ;`

Par contre, que l'OS utilisé soit MS Windows, linux ou Mac OS importe peu pour l'apprentissage de l'algorithmique et du développement en Java.

### 5.2.2 Écrire le code

Pour **écrire le code**, deux approches sont possibles : l'utilisation d'un éditeur de code ou d'un IDE (EDI, Environnement de Développement Intégré).

#### 1. éditeur de code

Un éditeur de code est un éditeur de texte — à ne pas confondre avec un traitement de texte — augmenté c'est-à-dire offrant des fonctionnalités supplémentaires. Citons par exemple :

- ▷ la coloration syntaxique. Le programme reconnaît les mots clés du langage, les structures et les écrit en couleur pour accroître la lisibilité du code ;
- ▷ l'indentation automatique et la réindentation du code ;

- ▷ une certaine autocomplétion pour les mots connus du langage et les variables.

Exemples d'éditeurs de code toutes plateformes, licences et prix confondus : *(g)Vim*, *Notepad++*, *Atom*, *SublimeText*...

Exemples d'éditeurs de texte inutiles pour le développement : *nano*, *Notepad*, *Mousepad*.

## 2. IDE

Un environnement de développement intégré est un programme servant à écrire les programmes. En plus d'un éditeur de code, un IDE compile en arrière plan, a un débogueur, organise les fichiers, crée des fichiers (en partie) pré-complétés sur base de *templates*...

Exemples d'IDE : *Netbeans*, *Eclipse*, *IntelliJ*...

Connaitre et utiliser correctement un éditeur de code et un IDE sont deux compétences essentielles d'un bon développeur. L'une n'allant pas sans l'autre.

### 5.2.3 Exécuter le programme

Quand le code est écrit, il faut le compiler puis l'exécuter. Les deux programmes, *javac* et *java*, font partie d'un ensemble de programmes fournis avec le langage Java. Cet ensemble de programmes nécessaires au développement en Java s'appelle **JDK** : *Java Development Kit*

Il en existe deux :

1. Java est — actuellement — la propriété d'Oracle et c'est Oracle qui fournit le JDK officiel.  
Télécharger Java chez Oracle <sup>43</sup>
2. OpenJDK est une alternative libre au JDK officiel Java.  
Télécharger OpenJDK <sup>44</sup>

**Remarque** Il ne faut pas confondre JDK et JRE. Un JRE pour *Java Runtime Environment*, est l'ensemble de programmes — il y en a moins — nécessaires à l'**exécution** de programmes Java. Dans ces programmes se trouve une machine virtuelle java (*JVM*) pour l'interprétation des programmes java.

Vous avez probablement déjà un — voire plusieurs — JRE sur votre machine.

### 5.2.4 Hello world

Une fois les armes fourbies, il est temps d'écrire son premier programme. Et, selon la tradition, nous allons écrire un programme qui affiche *Hello world*.

Un programme Java s'écrit dans une **classe**. Cette classe porte un nom et ce nom doit être le même que celui du fichier qui la contient. Nous allons appeler notre première classe **Hello**.

43. <http://www.oracle.com/technetwork/java/javase/downloads>

44. <http://openjdk.java.net/>

Grâce à mon éditeur de code, je crée le fichier `Hello.java`<sup>45</sup> qui contient :

```
public class Hello{  
2  public static void main (String[] args) {  
    System.out.println("Hello_world");  
}  
5 }
```

java

Je compile ma classe :

```
$  
javac Hello.java
```

terminal

Un fichier `Hello.class` apparaît dans mon répertoire courant. C'est le *bytecode* de ma classe. J'exécute ma classe :

```
$  
java Hello
```

terminal

... et je vois apparaître dans mon terminal les mots **Hello world**.

**Remarque** À partir de JDK11, voir [ ] ou au paragraphe 5.1.1 page 61, on peut se contenter d'une seule commande `java Hello.java`.

## 5.3 La grammaire du langage

Un langage de programmation, dès lors qu'il est compilé et exécuté par une machine, répond à des règles très strictes. En tout cas, ces règles doivent être non ambiguës.

Cet ensemble de règles est appelé la **grammaire du langage** et se trouve dans *The Java Language Specification*, un ouvrage reprenant toute la spécification du langage Java. Nous allons y faire référence régulièrement dans ces notes.

**JLS** Télécharger *The Java Language Specification*<sup>46</sup>

Les **symboles terminaux** de la grammaire, ceux que l'on peut retrouver en l'état dans le code sont écrit en police à chasse fixe comme ça. Les **règles de production**, qui sont définies ailleurs dans la grammaire, sont écrites en italiques comme ça.

Dans ces notes, nous utiliserons une **grammaire simplifiée** par soucis de simplification pour une première approche du développement sans jamais le préciser. Celles et ceux qui veulent aller plus loin sont invités à faire référence à JLS pour la grammaire complète.

45. Si vous êtes un utilisateur MS Windows, désactivez la propriété «*Hide extension for known files types*». Si vous ne le faites pas, vous allez — probablement — voir `Hello.java` alors que le fichier que vous créez est `Hello.java.txt`.

46. <https://docs.oracle.com/javase/specs/>

Nous avons dit que le type entier en Java était `int` et que les nombres réels étaient déclarés grâce au mot clé `double`. Si nous avions voulu utiliser la grammaire (simplifiée) nous aurions pu écrire :

```
NumericType:
    IntegralType
    FloatingPointType

IntegralType:
    int

FloatingPointType:
    double
```

Ce qui signifie qu'il existe deux sortes de types numériques : les nombres entiers et les nombres à virgule flottante. Pour les nombres entiers, il s'agit du type `int`. C'est un symbole terminal qui peut se retrouver tel quel dans un code. Pour les nombres réels — nous avons dit *pseudo-réels* — ou « à virgule flottante », il s'agit du type `double`.

Nous sommes incomplets à ce stade. Les personnes curieuses peuvent aller voir la grammaire complète de ces règles dans JLS10<sup>47</sup> p. 42.

---

47. <https://docs.oracle.com/javase/specs/jls/se10/jls10.pdf>



# Chapitre 6

## Une question de choix

Les **alternatives** permettent de n'exécuter des instructions que si une certaine *condition* est vérifiée. Par exemple, si le filtre à café est vide, le remplir. Dans la vie, nous testons notre environnement, dans nos algorithmes et dans nos programmes, nous allons tester les données.

Les algorithmes et les programmes vus jusqu'à présent ne proposent qu'un seul « chemin », une seule « histoire ». À chaque exécution de l'algorithme, les mêmes instructions s'exécutent dans le même ordre. Les alternatives permettent de créer des histoires différentes, d'adapter les instructions aux valeurs concrètes des données.

### Contenu

6.1	Le si ( <i>if-then</i> ) . . . . .	65
6.2	Le si-sinon ( <i>if-then-else</i> ) . . . . .	69
6.3	Le si-sinon-si . . . . .	72
6.4	Expression booléenne . . . . .	74
6.5	Le selon-que ( <i>switch</i> ) . . . . .	75

### 6.1 Le si (*if-then*)

Il existe des situations où des instructions ne doivent pas toujours être exécutées et un test va nous permettre de le savoir.

si une certaine condition est vraie alors  
 exécuter une ou plusieurs actions  
 fin si  
 continuer l'algorithme

*langage naturel*

Dans la grammaire du langage Java, cette instruction s'écrit :

*IfThenStatement:*

```
if ( Expression )  
    Statement
```

1. *Expression* représente une expression booléenne. c'est-à-dire ayant comme valeur **true** ou **false**.
2. *Statement* représente une instruction ou un **bloc** d'instructions. En langage Java, un bloc d'instructions est toujours délimités par une paire d'accolades.



### Remarque

Attention, un « si » n'est pas une règle que l'ordinateur doit apprendre et exécuter à chaque fois que l'occasion se présente. La condition n'est testée que lorsqu'on arrive à cet endroit de l'algorithme.

### Exemple

Supposons que la variable **nb** contienne un nombre positif ou négatif. Et supposons que l'on veuille le rendre positif. Il faudra tester son signe et, s'il est négatif, l'inverser. Par contre, s'il est positif, il n'y a rien à faire.

Voici comment écrire un algorithme sous différentes formes :

```
si nb < 0 alors  
    nb = -nb  
fin si
```

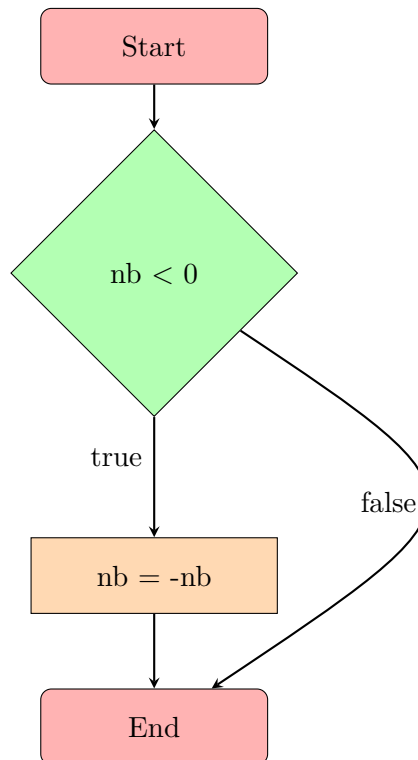
*langage naturel*

ou

```
if (nb < 0)  
    nb = -nb
```

*langage naturel*

Un organigramme aurait cette forme :



En langage Java, le *si* s'écrit comme suit :

```

1 if (nb < 0)
  nb = -nb;

```

java

Pour s'éviter des erreurs, nous utiliserons toujours le bloc d'instructions. Nous conseillons de faire de même et nous écrirons :

```

if (nb < 0){
  nb = -nb;
3 }

```

java

Traçons l'algorithme dans deux cas différents pour bien illustrer son déroulement.

#	nb	test
	-3	
1		vrai
2	3	

#	nb	test
	3	
1		faux

**Exercice de compréhension** Tracez l'algorithme écrit en langage Java avec les valeurs fournies et donnez la valeur de retour.

```
public static int exercice(int a, int b){  
2   int c;  
    c = 2 * a;  
    if (c > b){  
5       c = c-b;  
    }  
    return c;  
8 }
```

java

▷ exercice(2, 5) = \_\_\_\_\_

▷ exercice(4, 1) = \_\_\_\_\_

## 6.2 Le si-sinon (*if-then-else*)

La construction si-sinon permet d'exécuter certaines instructions ou d'autres en fonction d'un test.

```
si une certaine condition est vraie alors
    exécuter une ou plusieurs actions
sinon (la condition est alors fausse)
    exécuter d'autres actions
fin si
continuer l'algorithme
```

*langage naturel*

Dans la grammaire du langage Java, cette instruction s'écrit :

```
IfThenElseStatement:
    if ( Expression )
        Statement
    else
        Statement
```

### Exemple

Pour illustrer cette instruction, nous allons écrire un algorithme qui recherche le maximum de deux nombres.

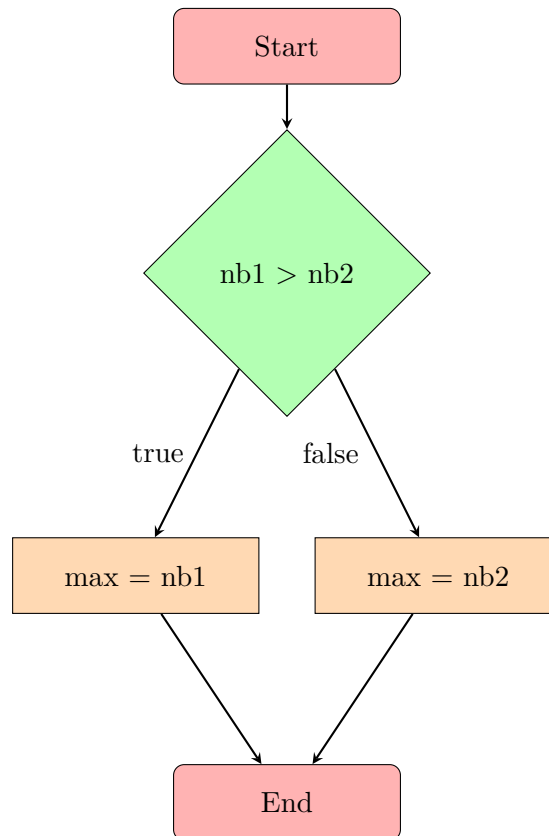
Pour déterminer le maximum de deux nombres, c'est-à-dire la plus grande des deux valeurs, il y aura deux chemins possibles. Le maximum devra prendre la valeur du premier nombre ou du second selon que le premier est plus grand que le second ou pas.

Voici comment écrire cet algorithme sous différentes formes :

```
if (nb1 > nb2)
    max = nb1
else
    max = nb2
```

*langage naturel*

Un organigramme aurait cette forme :



En supposant les variables déclarées, le *si-alors*, s'écrit comme suit en langage Java :

```

1  if (nb1 > nb2){
2      max = nb1;
3  } else {
4      max = nb2;
5  }
6

```

java

Traçons l'algorithme dans différentes situations.

#	nb1	nb2	max	test
	3	2	indéfini	
1			indéfini	vrai
2			3	

#	nb1	nb2	max	test
	4	42	indéfini	
1			indéfini	faux
4			42	

Le cas où les deux nombres sont égaux est également géré.

#	nb1	nb2	max	test
	4	4	indéfini	
1			indéfini	faux
4			4	

**Exercice de compréhension** Tracez ces algorithmes ou programmes avec les valeurs fournies et donnez la valeur de retour.

```

public static int exercice(int a, int b){
    int c;
3   if (a > b){
        c = a/b;
    } else {
6       c = b/a;
    }
9   }

```

java

▷ exercice(2, 3) = \_\_\_\_

▷ exercice(4, 1) = \_\_\_\_

```

public static int exercice(int x1, int x2){
    boolean ok;
3   ok = x1 > x2;
    if (ok){
        ok = ok && x1 == 4;
6   } else {
        ok = ok || x2 == 3;
    }
9   return x1 + x2;
}

```

java

▷ exercice(2, 3) = \_\_\_\_

▷ exercice(4, 1) = \_\_\_\_

### 6.3 Le si-sinon-si

Avec cette construction, il est possible d'indiquer à un endroit de l'algorithme plus de deux chemins possibles et dès lors que la première condition est fausse, tester une condition supplémentaire à chaque étape.

```
si une certaine condition est vraie alors
    exécuter une ou plusieurs actions
sinon si une autre condition est vraie alors
    exécuter d'autres actions
sinon
    exécuter encore d'autres actions
fin si
continuer l'algorithme
```

*langage naturel*

Dans la grammaire du langage Java, il n'y a pas d'instruction supplémentaire. Un *si-sinon-si* n'étant que des *si-sinon* imbriqués un peu comme suit :

```
if ( Expression )
    Statement
else
    if ( Expression )
        Statement
    else
        Statement
```

... qui seront généralement indentés comme suit :

```
if ( Expression )
    Statement
else if ( Expression )
    Statement
else
    Statement
```

... mais voyons cela sur un exemple.

#### Exemple

Supposons que l'on veuille mettre dans la chaîne `signe` la valeur "positif", "négatif" ou "nul" selon qu'un nombre donné est positif, négatif ou nul.

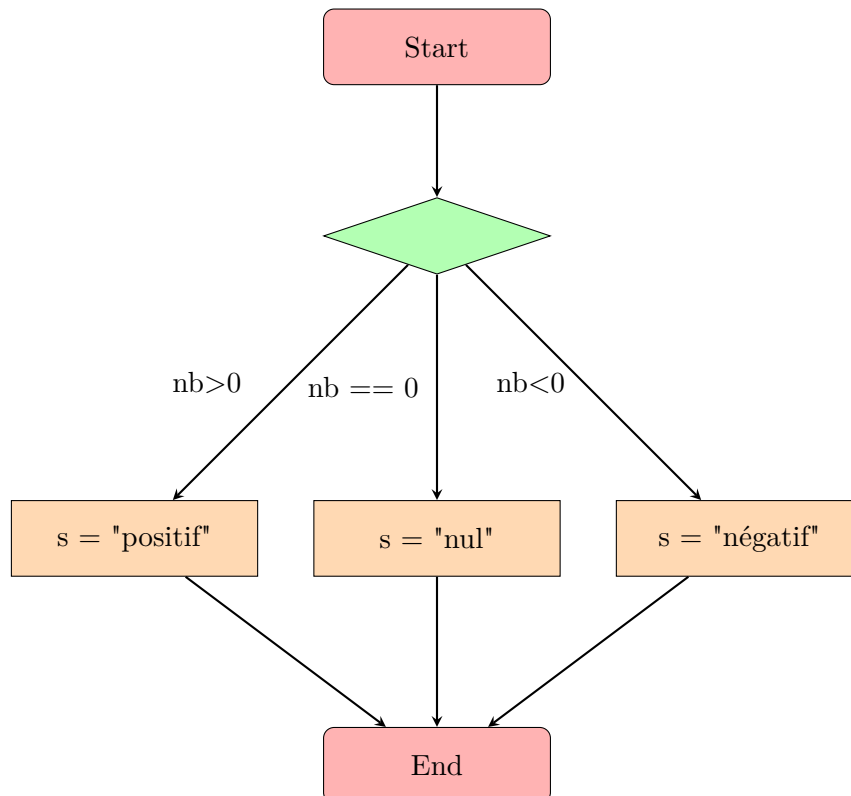
Voici comment écrire cet algorithme sous différents formats :



```
if (nb > 0)
    signe = "positif"
else if (nb == 0)
    signe = "nul"
else
    signe = "négatif"
```

*langage naturel*

Un organigramme aurait cette allure :



Comme dit plus haut, en langage Java, il n'y a pas de structure particulière pour ce test. Le if-then-else fait bien l'affaire. Seule l'indentation change un peu pour plus de lisibilité. Ce test peut donc s'écrire — en supposant toujours que les variables sont déclarées — comme ceci :

```
1 if (nb > 0){
    s = "positif";
} else if (nb == 0) {
4 s = "nul";
} else {
    s = "négatif";
7 }
```

*java*

Traçons l'algorithme dans différentes situations.

#	nb	signe	test
	2	indéfini	
1			vrai
2		"positif"	

#	nb	signe	test
	0	indéfini	
1			faux
3			vrai
4		"nul"	

#	nb	signe	test
	-5	indéfini	
1			faux
3			faux
6		"négatif"	

### Remarques.

- ▷ Pour le dernier cas, on se contente d'un **sinon** sans indiquer la condition ; ce serait inutile, elle serait toujours vraie.
- ▷ Le **si** et le **si-sinon** peuvent être vus comme des cas particuliers du **si-sinon-si**.
- ▷ On pourrait écrire la même chose avec des **si-sinon** imbriqués mais le **si-sinon-si** est plus lisible.
- ▷ Lorsqu'une condition est testée, on sait que toutes celles au-dessus se sont avérées fausses. Cela permet parfois de simplifier la condition.

**Exemple.** Supposons que le prix unitaire d'un produit (`prixUnitaire`) dépende de la quantité achetée (`quantité`). En dessous de 10 unités, on le paie 10€ l'unité. De 10 à 99 unités, on le paie 8€ l'unité. À partir de 100 unités, on paie 6€ l'unité.

```

1 if (quantité < 10){
    prixUnitaire = 10;
} else if (quantité < 100) {
4  // On sait que la quantité est plus grande
  // ou égale à 10. Inutile de tester.
    prixUnitaire = 8;
7 } else {
    prixUnitaire = 6;
}
10

```

java

## 6.4 Expression booléenne

Nous avons dit que dans un test, l'*expression* était une expression booléenne et nous avons vu quelques opérateurs intervenant dans ces expressions. Revenons plus en détail sur ce concept.



**Definition.** Une expression booléenne est une expression — c'est-à-dire le résultat d'un calcul — dont la valeur est booléenne : `true` ou `false`.

Une telle expression se compose grâce :

1. aux opérateurs relationnels (*relational operator* ou *comparators*);

Un opérateur relationnel est un opérateur dont la valeur est booléenne et les opérandes numériques.

*RelationalOperator:*  
(one of)  
< > <= >=

2. aux opérateurs d'égalité (*equality operators*);

Un opérateur d'égalité est un opérateur dont la valeur est booléenne et les opérandes de même type (à conversion près).

*EqualityOperator:*  
(one of)  
== !=

3. au complément logique (*logical complement operator*) et aux opérateurs conditionnels (*conditionals operators*);

Le complément logique et les opérateurs conditionnels sont des opérateurs dont la valeur est booléenne et le ou les opérandes également booléens.

Le && est prioritaire sur le ||.

*LogicalComplementOperator:*  
!  
*ConditionalOperator:*  
(one of)  
|| &&

## 6.5 Le selon-que (*switch*)

Cette nouvelle instruction permet d'écrire plus lisiblement *certaines* **si-sinon-si**, plus précisément quand le choix d'une branche dépend de la valeur précise d'une variable (ou d'une expression).

selon que la variable vaille :

- une valeur :  
    exécuter une ou plusieurs actions
- une autre valeur :  
    exécuter d'autres actions
- encore une autre valeur :  
    exécuter d'autres actions
- et une dernière valeur :  
    exécuter d'autres actions

fin selon que

*langage naturel*

Dans la grammaire du langage Java, cette instruction, *switch*, s'écrit (grammaire simplifiée) :

*SwitchStatement:*

`switch ( Expression ) SwitchBlock`

*SwitchBlock:*

*SwitchLabels Statement*

*SwitchLabel:*

`case ConstantExpression:`

`default:`

- ▷ l'expression ne peut pas être de n'importe quel type. À ce stade, elle peut être, un entier ou une chaîne et nous rencontrerons d'autres types plus tard ;
- ▷ *Statement* peut être une instruction ou plusieurs ;
- ▷ *SwitchLabels* (avec un *s*) ce sont plusieurs « **case** » ;
- ▷ le *switch* en java peut être vu comme un « *saut* au bon label ». Dès lors que l'instruction a trouvé le bon *case*, l'exécution des instructions continue. Il faut explicitement demander de sortir du *switch* en utilisant l'instruction **break** .

... mais voyons ça sur un exemple.

### Exemple.

Imaginons qu'une variable (*dayNumber*) contienne un numéro de jour de la semaine et qu'on veuille mettre dans une variable (*dayName*) le nom du jour correspondant ("lundi" pour 1, "mardi" pour 2...)

Une solution avec un **si-sinon-si** est possible mais le **selon-que** (*switch*) est plus lisible dans ce cas.

Voyons comment écrire un algorithme sous différentes formes :

```
switch ( dayNumber )
— 1 :
    dayName = "lundi"
— 2 :
    dayName = "mardi"
— 3 :
    dayName = "mercredi"
— 4 :
    dayName = "jeudi"
— 5 :
    dayName = "vendredi"
— 6 :
    dayName = "samedi"
— 7 :
    dayName = "dimanche"
```

*langage naturel*

En langage Java :

```
switch (dayNumber) {  
2  case 1:  
    dayName = "lundi";  
    break;  
5  case 2:  
    dayName = "mardi";  
    break;  
8  case 3:  
    dayName = "mercredi";  
    break;  
11 case 4:  
    dayName = "jeudi";  
    break;  
14 case 5:  
    dayName = "vendredi";  
    break;  
17 case 6:  
    dayName = "samedi";  
    break;  
20 case 7:  
    dayName = "dimanche";  
    }  
23
```

java

#### Remarques.

- ▷ Il peut y avoir plusieurs valeurs pour un cas donné.
- ▷ Il peut y avoir un cas par défaut, `default` qui sera exécuté si la valeur n'est pas reprise par ailleurs.



# Chapitre 7

## Module et références

### Contenu

7.1	Décomposer le problème . . . . .	<b>79</b>
7.2	Exemple . . . . .	<b>80</b>
7.3	Paramètres et valeur de retour . . . . .	<b>81</b>
7.3.1	Le paramètre en entrée . . . . .	81
7.3.2	Le paramètre en entrée-sortie . . . . .	82
7.3.3	La valeur de retour . . . . .	84
7.4	Type primitif et type référence . . . . .	<b>84</b>
7.4.1	Type primitif . . . . .	85
7.4.2	Type référence . . . . .	85
7.4.3	Les paramètres en Java . . . . .	86

### 7.1 Décomposer le problème

Jusqu'à présent, les problèmes que nous avons abordés étaient relativement petits. Nous avons pu les résoudre avec un algorithme d'un seul tenant.

Dans la réalité, les problèmes sont plus conséquents et il devient nécessaire de les décomposer en sous-problèmes. On parle d'une *approche modulaire*. Les avantages d'une telle décomposition sont multiples.

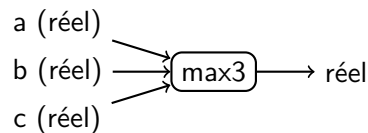
- ▷ **Cela permet de libérer l'esprit.** L'esprit humain ne peut pas traiter trop d'informations à la fois (*surcharge cognitive*). Lorsqu'un sous-problème est résolu, il peut se libérer l'esprit et attaquer un autre sous-problème.
- ▷ **On peut réutiliser ce qui a été fait.** Si un même sous-problème apparaît plusieurs fois dans un problème ou à travers plusieurs problèmes, il est plus efficace de le résoudre une fois et de réutiliser la solution.
- ▷ **On accroît la lisibilité.** Si, un algorithme, appelle un autre algorithme pour résoudre un sous-problème, le lecteur ou la lectrice verra un nom d'algorithme qui peut être plus parlant que les instructions qui se cachent derrière, même s'il y en a peu. Par exemple, `dizaine(nb)` est plus parlant que `nb MOD 100 DIV 10` pour calculer le nombre de dizaines d'un nombre.

Parmi les autres avantages, que vous pourrez moins percevoir en début d'apprentissage, citons la possibilité de répartir le travail dans une équipe.

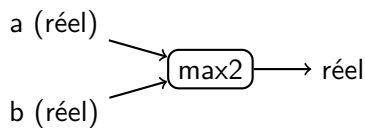
Un algorithme qui résout une partie de problème est parfois appelé **fonction**, **procédure**, **méthode** ou encore **module** en fonction du langage et du contexte. Il y a quelques nuances mais elles importent peu ici.

## 7.2 Exemple

Illustrons l'approche modulaire sur le calcul du maximum de 3 nombres.



Commençons par écrire la solution du problème plus simple : le maximum de 2 nombres.



```
1 public static double
    max2(double a, double b){
    double max;
    if (a > b){
4      max = a;
    } else {
    max = b;
7  }
    return max;
    }
10
```

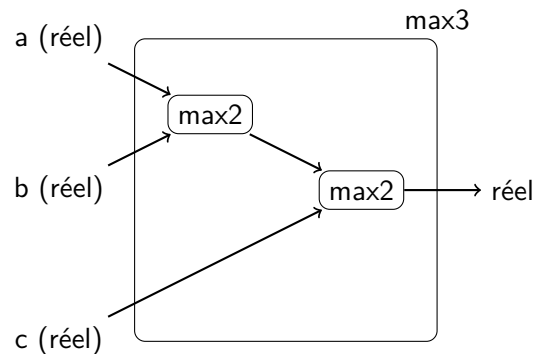
java

Pour le maximum de 3 nombres, il existe plusieurs approches. Voyons celle-ci :

1. Calculer le maximum des deux premiers nombres, soit **maxab**
2. Calculer le maximum de **maxab** et du troisième nombre, ce qui donne le résultat.



Elle s'illustre comme suit :



Sur base de cette idée, on voit que calculer le maximum de trois nombres peut se faire en calculant deux fois le maximum de deux nombres. On ne va évidemment pas *recopier*<sup>48</sup> dans notre solution ce qu'on a écrit pour le maximum de deux nombres; on va plutôt y faire référence, c'est-à-dire appeler l'algorithme `max2`. Ce qui donne :

```

public static double max3(double a, double b, double c){
2  double maxab, max;
   maxab = max2(a, b);
   max = max2(maxab, c);
5  return max;
}

```

java

qui peut se simplifier en :

```

public static double max3(double a, double b, double c){
2  return max2(max2(a, b), c);
}

```

java

## 7.3 Paramètres et valeur de retour

Jusqu'à présent, nous avons considéré que les paramètres d'un algorithme (ou *module*) correspondent à ses données et que le résultat, unique, est retourné.

Il s'agit d'une situation fréquente mais pas obligatoire que nous pouvons généraliser. De manière générale nous pouvons concevoir trois sortes de paramètres. Nous verrons ensuite ce qu'il en est en langage Java.

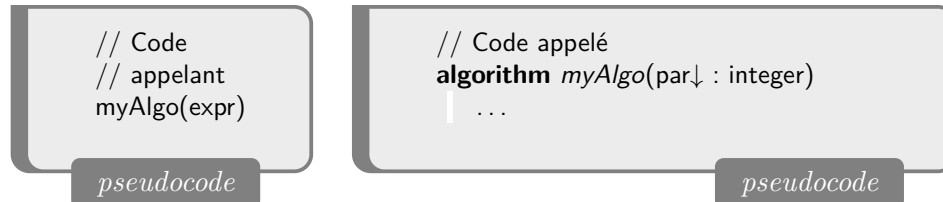
### 7.3.1 Le paramètre en entrée

Le paramètre en **entrée** est ce que nous connaissons déjà. Il correspond à une donnée de l'algorithme. Une valeur va lui être attribuée en début d'algorithme et elle ne sera pas modifiée. On pourrait faire suivre le nom du paramètre d'une flèche

48. Cette approche serait fastidieuse, engendrerait de nombreuses erreurs lors du recopiage et serait difficile à lire. Même le copier/coller n'est pas une bonne solution. Il diminue la lisibilité et rend la refactorisation et l'évolution des algorithmes et des programmes plus compliquées.

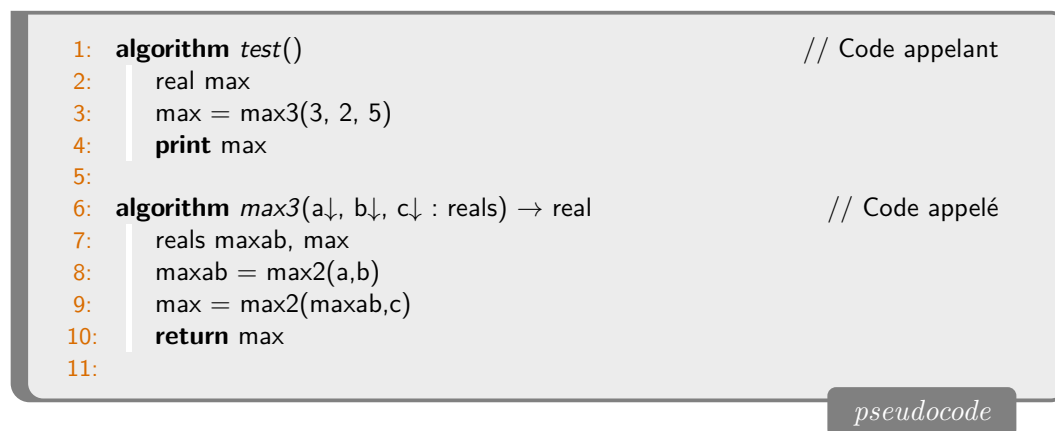
vers le bas ( $\downarrow$ ) pour rappeler son rôle mais ce n'est pas obligatoire lorsqu'il n'y a pas d'ambiguïté.

Lors de l'appel, c'est une **valeur** qui est fournie ou, plus généralement une expression dont la valeur sera donnée au paramètre. Voici un cas général de paramètre en entrée.



C'est comme si l'algorithme `myAlgo` commençait par l'affectation `par = expr`.

**Exemple.** Reprenons l'exemple de `max3` en ajoutant un petit test.



Traçons son exécution.

	test	max3				
#	max	a	b	c	maxab	max
2	indéfini					
3,7		3	2	5		
8					indéfini	indéfini
9					3	indéfini
10						5
11,3	5					

**Notez bien :** Dans cet exemple, on trouve deux fois la variable `max`. Il s'agit bien de deux variables **différentes** ; l'une est définie et connue dans `test` ; l'autre l'est dans `max3`.

### 7.3.2 Le paramètre en entrée-sortie

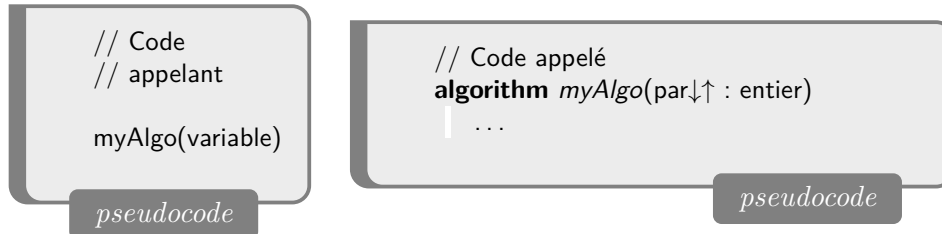
Le paramètre en **entrée-sortie** est un paramètre tel que :

- ▷ l'algorithme reçoit une valeur en entrée ;

▷ le paramètre peut être modifié.

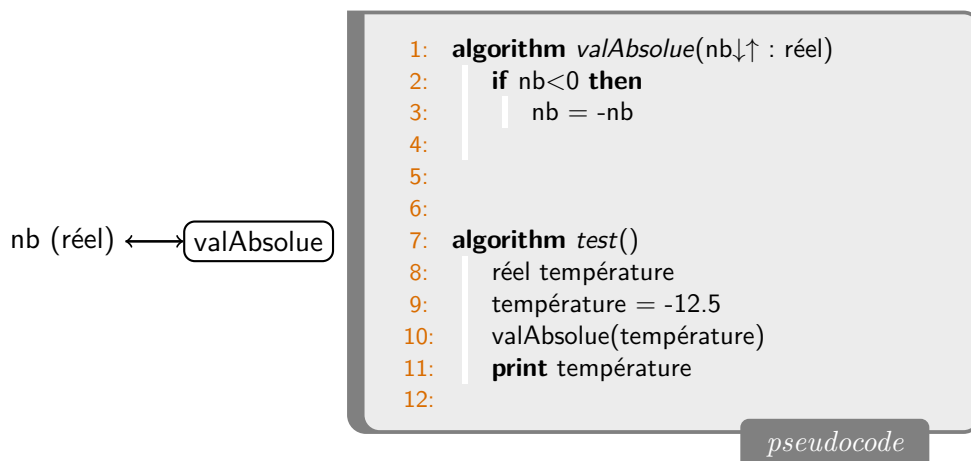
Cela signifie que l'algorithme a pour but de modifier le paramètre. Un tel paramètre pourrait être suivi d'une double flèche ( $\downarrow\uparrow$ ).

C'est une **une variable** qui doit être passée en paramètre. Sa valeur est donnée au paramètre au début de l'algorithme. À la fin de l'algorithme, la variable reçoit la valeur du paramètre. Voici un cas général de paramètre en sortie.



C'est comme si, dans le code appelé, il y avait une première ligne pour donner sa valeur au paramètre ( $\text{par} = \text{variable}$ ) et une dernière ligne pour effectuer l'assignation opposée ( $\text{variable} = \text{par}$ ). Il n'y a pas de **return**.

**Exemple.** Nous avons vu un algorithme qui retourne la valeur absolue d'un nombre. Nous pourrions imaginer une variante qui **modifie** le nombre reçu. En voici le schéma et la solution avec un appel possible :



Traçons-le.

	test	valAbsolue	
#	température	nb	test
8	indéfini		
9	-12.5		
10, 1		-12.5	
2			vrai
3		12.5	
5, 10	12.5		

**Remarque** Nous verrons dans la section 7.4 que le langage Java ne permet pas de traduire ces algorithmes en l'état.

### 7.3.3 La valeur de retour

La valeur de retour correspond au résultat de l'algorithme.

```
// Code  
// appelant  
integer var  
var = myAlgo()  
myAlgo()
```

*pseudocode*

```
// Code appelé  
algorithm myAlgo() → integer  
| ...
```

*pseudocode*

#### Remarques

- ▷ Cette valeur de retour est optionnelle, un algorithme peut ne rien retourner. Un algorithme qui ne **retourne** rien (pas de  $\rightarrow$ ) n'a pas de valeur ; il ne peut pas apparaître dans une expression ou être assigné à une variable.
- ▷ Le fait que la valeur de retour soit unique peut sembler rédhibitoire et c'est vrai. Ceci dit nous verrons qu'il existe plusieurs méthodes pour s'en sortir.

## 7.4 Type primitif et type référence

Dans nos algorithmes nous traitons avec des paramètres en entrée, en entrée/sortie et des valeurs de retour, qu'en est-il dans les langages de programmation ?

- ▷ Tous acceptent d'avoir une valeur de retour unique.
- ▷ Tous acceptent des paramètres en entrée.
- ▷ Certains et sous certaines conditions acceptent des paramètres en entrée/sortie.

Intéressons à la différence entre **type primitif** et **type référence**.

### 7.4.1 Type primitif

**Définition :** Une variable de type primitif est une variable qui contient directement la valeur qui lui est assignée. Cette variable a une taille fixe qui dépend de son type. L'emplacement mémoire qui lui est attribué se trouve sur la pile (*stack*<sup>49</sup>).



Par exemple, une variable de type `int` a une taille de 4 *bytes* (32 bits). Toujours.

Une variable `i` de type primitif et contenant la valeur 7 peut se représenter comme ci-contre.



Le langage Java accepte d'autres types primitifs, le détail se trouve dans le complément de la section 13 page 151.

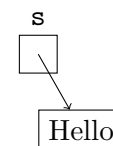
### 7.4.2 Type référence

**Définition :** Une variable de type référence est une variable qui ne contient pas directement la valeur qui lui est assignée. Elle contient une adresse mémoire désignant l'endroit où est — ou sera — stockée la valeur. L'emplacement mémoire attribué à la variable se trouve sur la pile et a la même taille pour toutes les variables de type référence tandis que l'emplacement mémoire qui contiendra effectivement la valeur sera attribué sur le tas (*heap*).



`String` est un type référence.

Une variable `s` de type référence et contenant la valeur "Hello" peut se représenter comme ci-contre.



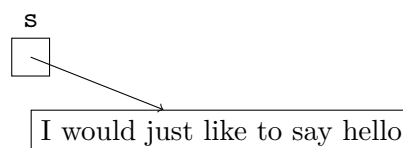
La même variable `s` peut recevoir une autre valeur, par exemple beaucoup plus grande I would just like to say hello.

```

String s = "Hello";
2  s = "I_would_just_like_to_say_hello";

```

java



**Remarque.** Même si nous ne connaissons actuellement qu'un seul type référence, ce seront les types les plus répandus.

Le langage Java accepte d'autres types références, le détail se trouve dans le complément de la section 13 page 151. Les tableaux sont aussi des types références comme nous le verrons plus tard dans la section 10 page 113.

49. Lorsqu'un programme s'exécute, le système lui attribue plusieurs emplacements mémoire : un contenant les instructions et deux qui contiendront les variables du programme. La pile (*stack*) et le tas (*heap*).

### 7.4.3 Les paramètres en Java

Dans nos algorithmes nous avons : une valeur de retour, des paramètres en entrée et des paramètres en entrée/sortie. L'ensemble étant optionnel. Qu'en est-il pour le langage Java ?

Tout comme en algorithmique, les méthodes Java permettent de retourner une valeur grâce au `return Expression` de fin de méthode. Une méthode peut ne rien retourner. Dans ce cas, il suffit de fermer l'accolade ouverte en début de méthode.

Les paramètres en Java se passent **par valeur**. En ce sens, c'est équivalent aux paramètres en entrée.

Traduisons l'exemple de la section 7.3.1 (p.81) en Java. Les valeurs 3, 2 et 5 sont passées en argument à la méthode `max3`. Les variables `a`, `b` et `c`, locales à la méthode `max3`, reçoivent les trois valeurs.

```
public class Test{
    public static void main(String[] args){
3      double max;
        max = max3(3,2,5);
        System.out.println("Maximum:␣" + max);
6    }

    public double max2(double a, double b){
9      //
    }

12   public double max3(double a, double b, double c){
        double maxab;
        double max;
15     maxab = max2(a,b);
        max = max2(maxab, c);
        return max;
18   }
}
```

java

Il n'y a pas de paramètre en entrée/sortie en Java puisque tous les passages de paramètres se font par valeur. Si le paramètre est de type référence, la valeur que reçoit la méthode est la valeur de la référence. Même s'il n'est pas possible de modifier le paramètre reçu, il sera possible de modifier l'objet ou le tableau **référéncé** par la valeur reçue en paramètre. En ce sens, c'est un peu un passage de paramètre en entrée/sortie.



**Remarque** Les paramètres déclarés dans l'entête d'un algorithme sont appelés **paramètres formels**. Les paramètres donnés à l'appel de l'algorithme sont appelés **paramètres effectifs**.

# Chapitre 8

## Un travail répétitif

Les ordinateurs révèlent tout leur potentiel dans leur capacité à répéter inlassablement les mêmes tâches. Vous avez pu appréhender les boucles si vous avez utilisé *studio.code.org* comme proposé dans la section 1.3 « Ressources » (cfr. p 14)

**Conseil pédagogique.** D'expérience, nous savons que ce chapitre est difficile à appréhender et qu'au fil des pages, la matière se complexifie. C'est en restant assidus et assidues; en faisant les exercices et en demandant un retour sur ses solutions aux enseignants et enseignantes que l'on met toutes les chances de son côté. Si l'on se sent perdu, il ne faut pas hésiter à demander de l'aide.



### Contenu

8.1	La notion de travail répétitif . . . . .	87
8.2	Une même instruction, des effets différents . . . . .	88
8.2.1	Exemple - Afficher les nombres de 1 à 5 . . . . .	88
8.3	Le « tant que » ( <i>while</i> ) . . . . .	90
8.3.1	Exemple - Afficher les nombres de 1 à 5 . . . . .	91
8.3.2	Exemple - Généralisation à n nombres . . . . .	91
8.4	Le « pour » ( <i>for</i> ) . . . . .	92
8.4.1	Exemples . . . . .	94
8.4.2	Un pas négatif . . . . .	95
8.4.3	Remarques . . . . .	95
8.4.4	Exemple – Afficher uniquement les nombres pairs . . . . .	95
8.5	« faire – tant que » . . . . .	96
8.5.1	Exemple . . . . .	97
8.6	Quel type de boucle choisir ? . . . . .	97
8.7	Exercices résolus . . . . .	98
8.7.1	Saisie de données multiples par l'utilisateur . . . . .	98
8.7.2	Lectures robustes . . . . .	101
8.7.3	Les suites . . . . .	102

### 8.1 La notion de travail répétitif

Lorsque l'on veut (faire) effectuer un travail répétitif, il faut indiquer deux choses :

- ▷ le travail à répéter ;
- ▷ quand s'arrêter ou quand continuer ou encore, combien de fois faire le travail.

Examinons quelques exemples pour fixer notre propos.

**Exemple 1.** Pour traiter des dossiers, nous pouvons dire « tant qu'il reste un dossier à traiter, le traiter » ou encore « traiter un dossier puis passer au suivant jusqu'à ce qu'il n'en reste plus à traiter ».

- ▷ La tâche à répéter est : « traiter un dossier ».
- ▷ Quand continuer : « s'il reste encore un dossier à traiter ».

Nous aurions pu dire de manière semblable :

- ▷ La tâche à répéter est : « traiter un dossier ».
- ▷ Quand s'arrêter : « dès qu'il n'y a plus de dossier à traiter ».

**Exemple 2.** Pour calculer la cote finale de tous les étudiants et toutes les étudiantes, nous dirions quelque chose du genre « Pour tout étudiant, calculer sa cote ».

- ▷ La tâche à répéter est : « calculer la cote d'un · e étudiant · e ».
- ▷ Combien de fois faire le travail : « autant qu'il y a d'étudiant · es »

Il faut le faire pour tous les étudiants et les étudiantes. Nous pourrions être plus précis et dire qu'il faut commencer au premier, passer à chaque fois au suivant et s'arrêter lorsque c'est terminé avec le dernier.

**Exemple 3.** Pour afficher tous les nombres de 1 à 100, nous dirions : « Pour tous les nombres de 1 à 100, afficher le nombre ».

- ▷ La tâche à répéter est : « afficher un nombre ».
- ▷ Nous indiquons qu'il faut le faire pour tous les nombres de 1 à 100. Il faut commencer à 1, passer au suivant et s'arrêter après avoir affiché 100.

## 8.2 Une même instruction, des effets différents

Comprenez bien que c'est toujours la même tâche qui est exécutée mais pas avec le même effet à chaque fois. Ainsi, c'est un dossier qui est traité mais à chaque fois un différent ; c'est un nombre qui est affiché mais à chaque fois un différent.

Par exemple, la tâche à répéter pour afficher des nombres ne peut pas être **afficher 1** ni **afficher 2** ni... Par contre, on pourra utiliser l'instruction **afficher nb** si on s'arrange pour que la variable **nb** s'adapte à chaque passage dans la boucle.

**De façon générale, pour obtenir un travail répétitif, il faut trouver une formulation de la tâche qui va produire un effet différent à chaque fois.**

### 8.2.1 Exemple - Afficher les nombres de 1 à 5

Si nous voulions un algorithme qui affiche les nombres de 1 à 5 sans utiliser de boucle, nous pourrions écrire :



```
print 1  
print 2  
print 3  
print 4  
print 5
```

*langage naturel*

Ces cinq instructions sont proches mais pas tout-à-fait identiques. En l'état, nous ne pouvons pas encore en faire une boucle<sup>50</sup> ; il va falloir ruser. Nous pouvons obtenir le même résultat avec l'algorithme suivant :

```
nb = 1  
print nb  
nb = 2  
print nb  
nb = 3  
print nb  
nb = 4  
print nb  
nb = 5  
print nb
```

*langage naturel*

ou encore

```
nb = 1  
print nb  
nb = nb + 1  
print nb  
nb = nb + 1  
print nb  
nb = nb + 1  
print nb  
nb = nb + 1  
print nb
```

*langage naturel*

Il est plus compliqué, mais cette fois les lignes 2 et 3 se répètent exactement. D'ailleurs, la dernière ligne ne sert à rien d'autre qu'à obtenir cinq copies identiques. Le travail à répéter est donc :

```
print nb  
nb = nb + 1
```

*langage naturel*

Cette tâche doit être effectuée cinq fois dans notre exemple. Il existe plusieurs structures répétitives qui vont se distinguer par la façon dont on va contrôler le nombre de répétitions. Voyons-les une à une<sup>51</sup>.

50. Vous vous dites peut-être que ce code est simple ; inutile d'en faire une boucle. Ce n'est qu'un exemple. Que feriez-vous s'il fallait afficher les nombres de 1 à 1000 ?

51. Nous ne verrons pas de structure de type **répéter 5 fois**... Elle est simple à comprendre mais pas souvent adaptée au problème à résoudre.

### 8.3 Le « tant que » (*while*)

Le « **tant que** » est une structure qui demande à l'exécutant de répéter une tâche (une ou plusieurs instructions) tant qu'une condition donnée est vraie.

tant que une certaine condition est vraie, répéter  
une ou plusieurs actions  
fin tant que  
continuer l'algorithme

*langage naturel*

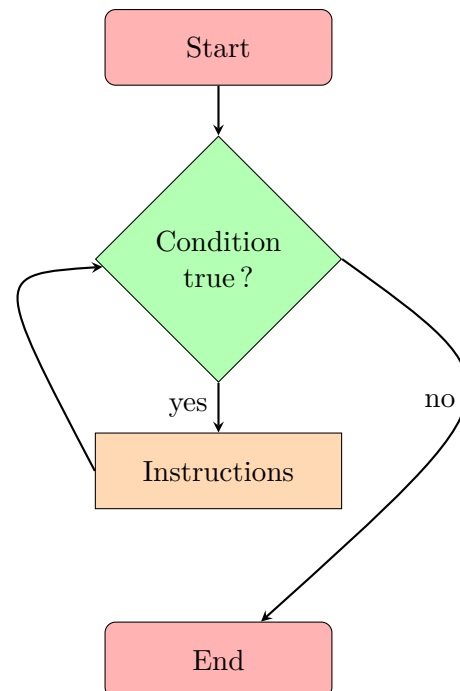
À la fin de chaque exécution des actions, la condition est retestée. Tant qu'elle est vraie les actions sont exécutées à nouveau.

Dans la grammaire du langage, cette instruction s'écrit :

*WhileStatement*  
**while** ( *Expression* )  
*Statement*

*Expression* est une expression booléenne comme nous l'avons vu (cf. section 6.4 p. 74) et *Statement* peut être une seule instruction ou un bloc d'instructions délimitées par des accolades.

L'organigramme ci-contre décrit le déroulement de cette structure. On remarquera que si la condition est fausse dès le début, la tâche n'est jamais exécutée.



#### Remarques

Comme pour la structure if, la **condition** est une expression à valeur booléenne.

Dans ce type de structure, il faut qu'il y ait dans la séquence d'instructions du **while** c'est-à-dire le bloc indenté au moins une instruction qui modifie une des composantes de la condition de telle manière qu'elle puisse devenir **fausse** à un moment donné.

Dans le cas contraire, la condition reste indéfiniment vraie et la boucle va tourner sans fin, c'est ce qu'on appelle une **boucle infinie**.

### 8.3.1 Exemple - Afficher les nombres de 1 à 5

Reprenons notre exemple d'affichage des nombres de 1 à 5. Pour rappel, la tâche à répéter est :

```
print nb
nb = nb + 1
```

*langage naturel*

La condition va se baser sur la valeur de **nb**. On continue tant que le nombre n'a pas dépassé 5. Ce qui donne (en n'oubliant pas l'initialisation de **nb**) :

```
1 public static void count5(){
    int nb = 1;
    while (nb <= 5){
4      System.out.println(nb);
      nb = nb + 1;
    }
7    System.out.println(
      "nb_vaut:" + nb);
    }
10
```

java

#	nb	condition	affichage
2	indéfini		
3	1		
4		vrai	
5			1
6	2		
4		vrai	
5			2
6	3		
4		vrai	
5			3
6	4		
4		vrai	
5			4
6	5		
4		vrai	
5			5
6	6		
4		faux	
8			nb vaut 6

### 8.3.2 Exemple - Généralisation à n nombres

Généraliser l'algorithme pour qu'il reçoive le nombre d'itérations en paramètres se fait simplement comme suit :

```
public static void count(int n){  
2   int i = 1;  
   while (i <= n){  
       System.out.println(i);  
5       i = i + 1;  
   }  
   System.out.println(  
8       "i_vaut: " + i);  
}
```

java

## 8.4 Le « pour » (*for*)

Cette structure va plutôt indiquer **combien de fois** la tâche doit être répétée. Cela se fait au travers d'une **variable de contrôle** dont la valeur va évoluer à partir d'une valeur de départ jusqu'à une valeur finale.

pour une variable allant d'une valeur de départ à une valeur finale  
exécuter une ou plusieurs actions  
fin pour  
continuer l'algorithme

*langage naturel*

Ou, en étant un peu plus précis et en précisant un pas différent de 1 :

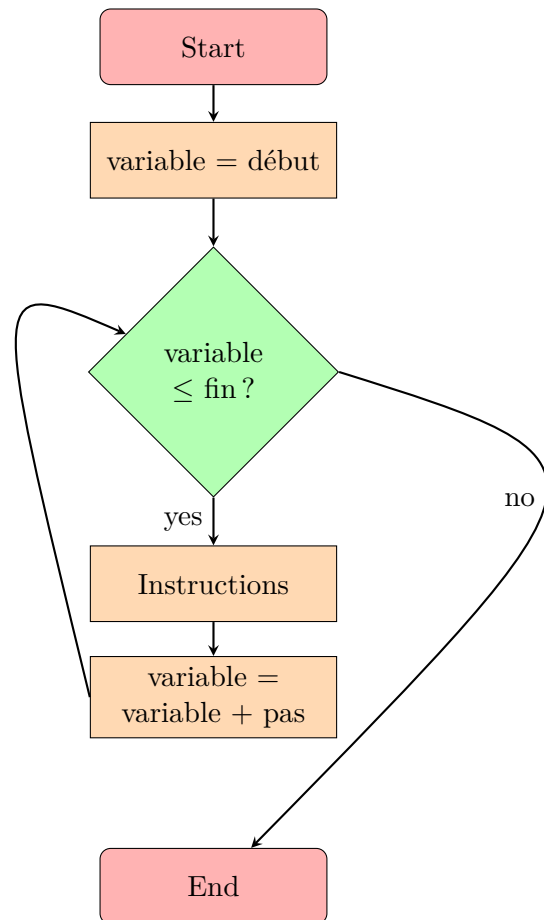
pour variable = début jusqu'à fin par pas  
exécuter une ou plusieurs actions  
fin pour  
continuer l'algorithme

*langage naturel*

Dans ce type de structure, **début**, **fin** et **pas** peuvent être des constantes, des variables ou des expressions entières.

La boucle s'arrête lorsque la variable dépasse la valeur de fin.

L'organigramme ci-contre illustre le fonctionnement d'une boucle for.



Dans la grammaire du langage Java, l'instruction for est un peu plus large que de dire qu'une variable parcourt les valeurs allant de **début** à **fin**. Voici sa grammaire (simplifiée).

*ForStatement:*

```
for ( ForInit; Expression; ForUpdate )
    Statement
```

#### **ForInit**

initialise la variable ;

Par exemple `variable = début`.

Il sera nécessaire de la déclarer.

#### **Expression**

est la condition de continuation ;

#### **ForUpdate**

est l'incrément, le passage à l'itération suivante ;

Par exemple  $\text{variable} = \text{variable} + \text{pas}$

Une utilisation simple et fréquente du `pour` a cette forme :

```
for (int i = début; i < fin; i = i + pas){
2  // statement
}
```

java

Nous verrons plus loin des exemples plus généraux du `for` et nous verrons également qu'il existe également un *enhanced for* — appelé *foreach* — permettant une écriture très simple pour le parcours de certaines collections.

### 8.4.1 Exemples

Reprenons notre exemple d'affichage des nombres de 1 à 5. Voici la solution avec un `for` et le traçage correspondant.

```
public static void countTo5(){
2  for (int nb=1; nb <= 5; nb =
    nb+1){
    System.out.println(i);
  }
5  System.out.println("nb_n'existe_
    plus");
}
```

java

#	nb	cond.	affichage
3	1	vrai	
4			1
3	2	vrai	
4			2
3	3	vrai	
4			3
3	4	vrai	
4			4
3	5	vrai	
4			5
3	6	faux	
6	#	#	nb n'existe plus

Pour généraliser à  $n$  nombres, nous pouvons écrire :

```
public static void countToN(int n){
2  for (int i = 1; i <= n; i = i + 1){
    System.out.println(i);
  }
5 }
```

java

**Remarques**

- ▷ En langage Java, l'indice de la boucle est souvent noté *i*.
- ▷ Lorsqu'il n'est pas explicitement demandé le contraire, l'indice d'une boucle commence à 0.

**8.4.2 Un pas négatif**

Le pas est parfois négatif, dans le cas d'un compte à rebours, par exemple. Dans ce cas, la boucle s'arrête lorsque la variable prend une valeur plus petite que la valeur de fin et il faut faire attention à bien formuler la condition.

**Exemple :** Compte à rebours à partir de *n*.

```
public static void countTo1(int n){  
    for (int i = n; i > 0; i = i - 1){  
3        System.out.print(i);  
    }  
    System.out.println("Go_!");  
6 }
```

java

**8.4.3 Remarques**

1. Il faut veiller à la cohérence de cette structure.

Nous pouvons considérer pour nos algorithmes qu'au cas (à éviter) où **début** est strictement supérieur à **fin** et le **pas** est positif, la séquence d'instructions n'est jamais exécutée (mais ce n'est pas le cas dans tous les langages de programmation!). Idem si **début** est strictement inférieur à **fin** mais avec un **pas** négatif.

2. Il est important de ne pas modifier les variables de contrôle **début**, **fin** ou **pas** ni de modifier la **variable** au sein de la boucle pour garder une boucle lisible.

**8.4.4 Exemple – Afficher uniquement les nombres pairs**

Cette fois-ci, nous allons écrire une boucle affichant uniquement les nombres **pairs** jusqu'à la limite *n*.

**Exemple :** Les nombres pairs de 1 à 10 sont : 2, 4, 6, 8, 10.

Notez que *n* peut être impair. Si *n* vaut 11, l'affichage est le même que pour 10. Avec un « pour », une solution est :

```
public static void printEven(int n){  
    for (int i=2; i <= n; i = i + 2){  
3        System.out.println(i);  
    }  
    }  
6 }
```

java

## 8.5 « faire – tant que »

À l'inverse du « tant que - faire » qui peut ne pas exécuter la séquence d'instructions si le test est faux dès le départ, il existe une structure qui exécutera au moins une fois la séquence d'instructions avant de tester la condition. Cette instruction peut-être :

- ▷ « faire - jusqu'à ce que » ou ;
- ▷ « faire - tant que ».

Cette structure est très proche du «tant que - faire » à ceci près que le test est fait à la fin et pas au début.

faire  
une ou plusieurs actions  
jusqu'à ce qu'une certaine condition soit vraie  
continuer l'algorithme

*langage naturel*

ou

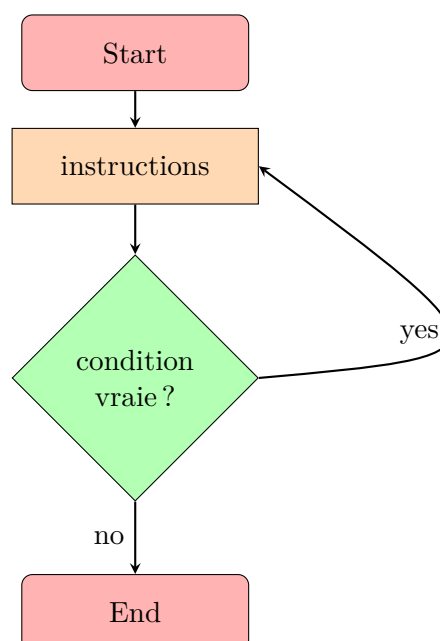
faire  
une ou plusieurs actions  
tant que une certaine condition est vraie  
continuer l'algorithme

*langage naturel*

En Java, c'est le second choix qui est retenu

```
DoStatement:
do
    Statement
while ( Expression );
```

Comme avec le tant-que, il faut que la séquence d'instructions comprise entre **faire** et **tant que** contienne au moins une instruction qui modifie la condition de telle manière qu'elle puisse devenir **vraie** à un moment donné pour arrêter l'itération. Le schéma ci-contre décrit le déroulement de cette boucle.





### 8.5.1 Exemple

Reprenons notre exemple d’affichage des nombres de 1 à 5. Voici la solution et le traçage correspondant.

#	nb	condition	affichage
2	indéfini		
3	1		
5			1
6	2		
7		vrai	
5			2
6	3		
7		vrai	
5			3
6	4		
7		vrai	
5			4
6	5		
7		vrai	
5			5
6	6		
7		faux	
8			nb vaut 6

```

public static void count5(){
    int nb = 1;
3   do {
        System.out.println(nb);
        nb = nb + 1;
6   } while (nb <= 5);
    System.out.println("nb_
        vaut_" + nb);
9   }

```

java

**Exercice** Tracer l’algorithme si l’instruction 2 est `int nb = 10;`.

## 8.6 Quel type de boucle choisir ?

En pratique, il est possible d’utiliser systématiquement la boucle **tant que** qui peut s’adapter à toutes les situations.

Cependant, il est plus clair d’utiliser la boucle **pour** dans les cas où le nombre d’itérations est fixé et connu à l’avance. Par là, nous voulons dire que le nombre d’itérations est déterminé au début de la boucle.

La boucle **faire-tant que** convient quant à elle dans les cas où le contenu de la boucle doit être parcouru au moins une fois, alors que dans **tant que**, le nombre de parcours peut être nul si la condition initiale est fausse.

La figure fig.8.1 ci-dessous propose un récapitulatif.

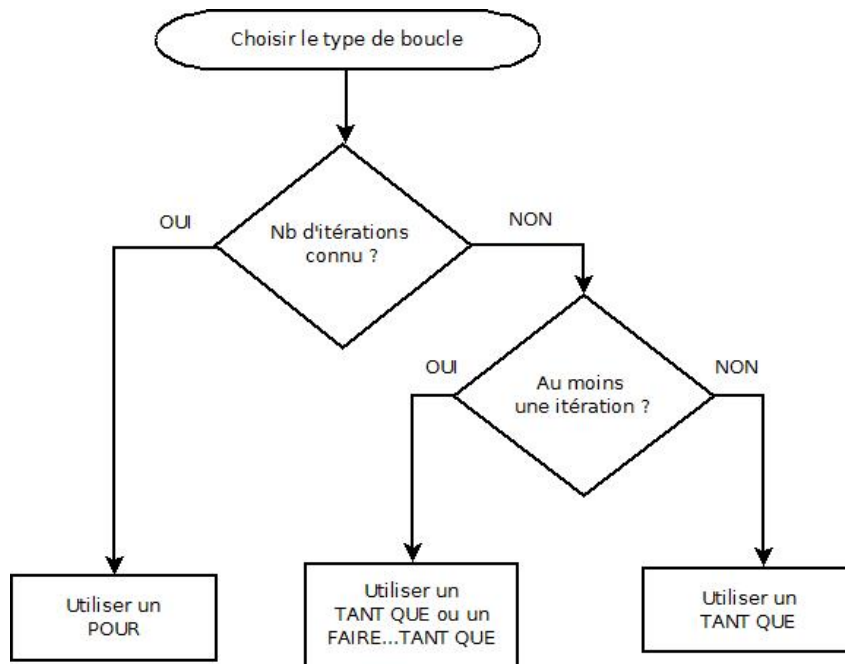


FIGURE 8.1 – Récapitulatif, choix d’une structure répétitive

## 8.7 Exercices résolus

### 8.7.1 Saisie de données multiples par l’utilisateur

Il existe des problèmes où l’algorithme doit demander une série de valeurs à l’utilisateur pour pouvoir les traiter. Par exemple, les sommer, en faire la moyenne, calculer la plus grande...

Dans ce genre de problème, l’algorithme ou le programme stocke chaque valeur donnée par l’utilisateur dans une seule et même variable et la traite avant de passer à la suivante. Prenons un exemple concret pour mieux comprendre.

Écrire un algorithme et un programme qui calcule et retourne la somme d’une série de nombres donnés par l’utilisateur.

Il faut d’abord se demander comment l’utilisateur va pouvoir indiquer combien de nombres il faut additionner ou quand est-ce que le dernier nombre à additionner a été entré. Voyons quelques possibilités.

#### 8.7.1.1 Variante 1 : le nombre de valeurs est connu

L’utilisateur indique le nombre de termes au départ. Ce problème est proche de ce qui a déjà été fait.

En langage Java, l’habitude est de commencer à compter à partir de 0. Là où nous avons l’habitude de compter de 1 à  $n$ , nous compterons toujours de 0 à  $n - 1$ . Plutôt que d’avoir comme condition de fin  $i \leq n - 1$ , nous écrirons  $i < n$ .

Une solution en Java peut-être :

```

public static int sum(){
    int nValues;    // Nombre de valeurs à additionner
3   int value;      // Terme de l'addition
    int sum = 0;    // Somme construite au fur et à mesure
    Scanner keyboard = new Scanner(System.in); // nécessite un import
6
    nValues = keyboard.nextInt();
    for (int i = 0; i < nValues; i = i + 1){
9        value = keyboard.nextInt();
        sum = sum + value;
    }
12   return sum;
}

```

java

**Remarque** Dans les exemples, nous utilisons des lectures au clavier très rudimentaires. Par exemple :

```

1 value = keyboard.nextInt();

```

java

Cette instruction, lorsqu'elle est exécutée, se traduit par un curseur qui clignote en attendant un nombre entier. Ce serait mieux de demander à l'utilisateur au préalable d'entrer un entier en lui écrivant un message. Plutôt comme suit :

```

1 System.out.print("Entrer un entier: ");
  value = keyboard.nextInt();

```

java

`nextInt` lit un entier au clavier. Si ce n'est pas un entier qui est entré... une erreur survient et le programme plante lamentablement. Pour éviter ceci, nous pouvons demander au programme de vérifier que l'utilisateur ou l'utilisatrice entre bien un entier, sinon, lui signaler. Justement la classe `Scanner` propose une méthode `hasNextInt`. Nous pourrions alors écrire ceci :

```

    System.out.print("Entrer un entier: ");
    while (!keyboard.hasNextInt()){
3        keyboard.next();
    }
    value = keyboard.nextInt();
6

```

java

Lors de l'appel à la méthode `hasNextInt`, l'utilisateur va devoir entrer une valeur qui sera mémorisée (*bufferisée*) et la méthode retourne `true` si l'utilisateur a entré un entier, `false` sinon.

- ▷ Si c'est `true`, c'est un entier. Le programme n'entre pas dans la boucle et se rend directement à l'instruction suivante. Cette instruction consomme la valeur lue précédemment et la place dans la variable `value`.

- ▷ Si c'est `false`, ce n'est pas un entier. Le programme entre dans la boucle et l'instruction `keyboard.next()` consomme la valeur lue précédemment... et n'en fait rien. Elle vide simplement le *buffer*.

Le programme retourne au test et réexécute `hasNextInt` qui lit une valeur au clavier et la place dans le *buffer*. Et ainsi de suite.

### 8.7.1.2 Variante 2 : stop ou encore

Après chaque nombre, l'algorithme ou le programme demande à l'utilisateur s'il y a encore un nombre à additionner.

Dans ce cas, il faut chercher une solution différente car le nombre de valeurs à additionner — et donc le nombre d'exécution de la boucle — n'est pas connu au départ. Il faudra utiliser un « tant que » ou un « faire-tant que ».

Si nous demandons en fin de boucle s'il reste encore un nombre à additionner l'algorithme aura l'allure suivante où nous utilisons un « do - while » :

```
public static int sum(){
    /* Est-ce qu'il reste une variable à additionner (hasMore) */
3   boolean hasMore;
    int value;
    int sum = 0;
6   Scanner keyboard = new Scanner(System.in); // nécessite un import

    do {
9       value = keyboard.nextInt();
        sum = sum + value;
        hasMore = keyboard.nextBoolean();
12  } while (hasMore);
    return sum;
}
15
```

java

Avec cette solution, on additionne au moins une valeur. Il est possible de tenir compte du cas particulier où l'utilisateur ne veut additionner aucune valeur. Dans ce cas, il faut utiliser un « tant que » et donc poser la question avant d'entrer dans la boucle.

```
public static int sum(){
    /* Est-ce qu'il reste une variable à additionner (hasMore) */
3   boolean hasMore;
    int value;
    int sum = 0;
6   Scanner keyboard = new Scanner(System.in); // nécessite un import

    hasMore = keyboard.nextBoolean();
9   while (hasMore){
        value = keyboard.nextInt();
        sum = sum + value;
12  hasMore = keyboard.nextBoolean();
    }
    return sum;
15 }
```

java

### 8.7.1.3 Variante 3 : valeur sentinelle

L'utilisateur entre une valeur spéciale pour indiquer la fin. Il s'agit d'une valeur **sentinelle**.

Cette solution n'est envisageable que si cette valeur **sentinelle** ne fait pas partie de l'ensemble des valeurs possibles.

Dans notre exemple si on veut additionner des nombres positifs uniquement, la valeur -1 peut servir de valeur sentinelle. Mais sans limite sur les nombres à additionner (positifs, négatifs ou nuls), il n'est pas possible de choisir une sentinelle.

Ici, on se base sur la valeur entrée pour décider si on continue ou pas. Il faut donc **toujours** effectuer un test après une lecture de valeur. C'est pour cela qu'il faut effectuer une lecture avant la boucle et une autre à la fin de la boucle.

```
public static int sum(){
2   int value;          // Un des termes de l'addition
    int sum = 0;        // La somme
    Scanner keyboard = new Scanner(System.in); // nécessite un import
5
    value = keyboard.nextInt();
    while (value != -1){
8        sum = sum + value;
        value = keyboard.nextInt();
    }
11   return sum;
}
```

java

## 8.7.2 Lectures robustes

Jusqu'à présent pour lire un entier au clavier on utilisait un code tel que :

```
Scanner keyboard = new Scanner(System.in); // nécessite un import
2 int i = keyboard.nextInt();
```

java

Cependant cette technique a le désavantage de produire une erreur si l'utilisateur ou l'utilisatrice met autre chose qu'un entier (du texte, une virgule, etc.), de sorte que le programme s'arrête net. Pour pallier à cela, nous voudrions un algorithme qui remplacerait `keyboard.nextInt()`. Cet algorithme devrait faire :

tant qu'un entier n'a pas été entré :  
(re-)demande un entier à l'utilisateur  
utiliser l'entier entré.

*langage naturel*

L'API Java offre à cette fin une autre méthode : `keyboard.hasNextInt()` qui vérifie si la personne utilisatrice a effectivement entré un entier. Notre algorithme peut alors s'écrire en Java :

```
1 Scanner keyboard = new Scanner(System.in); // nécessite un import
  System.out.print("Entrez un entier:");
  while (! keyboard.hasNextInt()) {
4     keyboard.next();
    System.out.print("Ce n'est pas un entier! Ré-essayez:");
  }
7 int reponse = keyboard.nextInt();
  // maintenant on peut utiliser cet entier.
```

java

Pour bien comprendre cet algorithme, il faut noter que :

- ▷ `hasNextInt()` met le programme en pause en attendant que l'utilisateur entre un *token* (un mot ou un nombre) sur l'entrée standard (une sorte de file d'attente des choses écrites au clavier);
- ▷ si le token entré est un entier alors la méthode retourne `true`, sinon elle retourne `false` – rien de plus.
- ▷ `next()` attend aussi qu'un token soit entré... la différence est que `next` va enlever le token de l'entrée standard et le retourner. Comme ce n'était pas un entier (`hasNextInt` nous l'a dit), on le jette;
- ▷ en sortie de boucle on sait que le prochain token sur l'entrée standard est un entier;
- ▷ la méthode `nextInt()` lit ce token et le convertit en nombre pour nous.

### 8.7.3 Les suites

Nous avons vu quelques exemples d'algorithmes qui affichent une suite de nombres (par exemple, afficher les nombres pairs). Nous avons pu les résoudre facilement avec un **for** en choisissant judicieusement les valeurs de début et de fin ainsi que le pas.

Ce n'est pas toujours aussi simple. Nous allons voir deux exemples plus complexes et les solutions qui les accompagnent. Elles pourront se généraliser à beaucoup d'autres exemples.

**8.7.3.1 Exemple 1 - Afficher les carrés**

Nous voulons afficher les  $n$  premiers nombres carrés parfaits : 1, 4, 9, 16, 25...

Si l'on veut savoir quel est le 7<sup>e</sup> nombre à afficher, c'est  $7^2$ , soit 49. Plus généralement, le nombre à afficher lors du  $i^e$  passage dans la boucle est  $i^2$ .

étape	1	2	3	4	5	6	7	8
valeur à afficher	1	4	9	16	25	36	49	64

L'algorithme qui en découle est :

```
1 public static void squarres(int n){
    for (int i = 1; i <= n; i++){
        System.out.println(Math.pow(i, 2));
    }
}
```

java

Dans cette solution, la variable de contrôle compte simplement le nombre d'itérations. Le calcul du nombre à afficher se fait en fonction de cette variable de contrôle (ici le carré convient). Par une vieille habitude des programmeurs<sup>52</sup>, une variable de contrôle qui se contente de compter les passages dans la boucle est souvent nommée  $i$ . Elle est appelée « indice » de parcours de la boucle.

Cette solution peut être utilisée chaque fois qu'on peut calculer le nombre à afficher en fonction de  $i$ .

Cet exemple illustre un premier modèle important d'algorithme de génération de suite, modèle dans lequel l'élément à la  $i^e$  place, souvent appelé le  $i^e$  terme, peut être déterminé directement au départ d'une fonction de l'indice  $i$ .

$$i \longrightarrow f(i)$$

**8.7.3.2 Exemple 2 - Une suite un peu plus complexe**

Écrire un algorithme qui affiche les  $n$  premiers nombres de la suite : 1, 2, 4, 7, 11, 16...

Comme nous pouvons le constater, à chaque étape on ajoute un peu plus au nombre précédent.

$$1 \xrightarrow{+1} 2 \xrightarrow{+2} 4 \xrightarrow{+3} 7 \xrightarrow{+4} 11 \xrightarrow{+5} 16 \dots$$

Ici, difficile de partir de la solution de l'exemple précédent car il n'est pas facile de trouver la fonction  $f(i)$  qui permet de calculer le nombre à afficher en fonction de  $i$ .

Par contre, il est assez simple de calculer ce nombre en fonction du précédent.

$$\text{nb à afficher} = \text{nb affiché juste avant} + i$$

<sup>52</sup>. Née avec le langage FORTRAN où la variable  $i$  était par défaut une variable entière.

Sauf pour le premier, qui ne peut pas être calculé en fonction du précédent.

Mathématiquement, cette suite s'écrirait

$$x_0 = 1$$
$$x_i = x_{i-1} + i$$

Une solution élégante et facilement adaptable à d'autres situations est :

```
value = première valeur à afficher
for i de 1 à n
  print value
  value = valeur suivante calculée à partir de la précédente
```

*langage naturel*

qui, dans notre exemple précis, devient :

```
public static void suite(int n){
    int value = 1;
3   for (int i=1; i <= n; i = i + 1){
        System.out.println(value);
        value = value + i;
6   }
}
```

java

Cet exemple illustre un second modèle de génération de suite dans lequel l'élément à la  $i^e$  place, le  $i^e$  terme, est exprimé en fonction du ou des précédents termes. Ce genre de suite, s'appelle « suite récurrente ».



# Chapitre 9

## Les tests

*« J'ai fait tourner le programme, il fonctionne bien. »*

Tous les programmes contiennent des erreurs, des *bugs*, des défauts... Ces « *bugs* » entraînent, au mieux, un inconfort pour l'utilisateur ou l'utilisatrice. Ils peuvent occasionner une perte de temps, d'argent, de données, de matériel... ou, pire, un danger pour la vie humaine dans un environnement industriel par exemple.

Le travail de développeur ou de la développeuse est de réduire le nombre d'erreurs dans son programme. Pour ce faire, il ou elle le **testera** abondamment. Ce chapitre s'intéresse aux tests.

### Contenu

9.1	Les types d'erreurs et de tests . . . . .	<b>105</b>
9.2	Planifier les tests . . . . .	<b>106</b>
9.3	JUnit . . . . .	<b>107</b>
9.4	Exemple . . . . .	<b>109</b>

## 9.1 Les types d'erreurs et de tests

Les défauts d'un programme peuvent être de différentes sortes :

1. les erreurs de compilation ;  
Ces erreurs sont détectées par le compilateur et rapidement corrigées.
2. les erreurs d'exécution  
... et le programme s'arrête ;  
... et le programme ne fournit pas les bonnes valeurs ;  
Ces erreurs sont difficiles à corriger. Elles nécessitent que le programme soit **testé** avec un nombre suffisant de situations différentes. Elles peuvent être dues à des erreurs de programmation mais aussi à des erreurs dans nos algorithmes.

3. les erreurs de programmation qui entraînent une lenteur du programme ou une consommation excessive de mémoire.

Pour éviter ces erreurs, respecter les bonnes pratiques de développement est une première étape qu'il ne faut pas négliger. Ensuite, il faudra détecter les endroits dans le code où les ralentissements ont lieu ou la consommation de mémoire augmente. Il existe des logiciels dédiés à ce genre de cas.

Pour montrer — garantir, prouver — qu'un programme fonctionne bien, il ne suffit pas de montrer qu'il fournit les bons résultats dans certains cas, il faut convaincre, qu'il fournira les bons résultats dans **tous** les cas :

- ▷ les cas généraux ;
- ▷ les cas particuliers ;
- ▷ lors d'une défaillance de l'environnement ;

Par exemple, si un capteur de distance est défectueux, il faut que la chaîne de production s'arrête.

- ▷ lors d'une utilisation non conforme du programme.

Pour répondre à ces attentes, le développeur ou la développeuse sera attentive<sup>53</sup> à utiliser une méthodologie éprouvée dans toutes les étapes de développement de son projet ; de la phase d'analyse au déploiement. Il mettra en place différents types de tests<sup>54</sup> ; des tests unitaires, d'intégration, fonctionnels, de non-régression. . . Dans ce premier cours de développement, nous nous intéressons aux **tests unitaires**. Pour ces tests unitaires, le développeur ou la développeuse fournira un ensemble de valeurs à tester représentatives et convaincantes.



### Définition

**Test unitaire** Procédure permettant de tester le bon fonctionnement d'une unité de code, une méthode JAVA dans notre cas. En fonction des paramètres qu'elle reçoit en entrée, la méthode fournit-elle les bons résultats ?

L'idée sous-jacente des tests unitaires est que si chaque partie est correcte, l'ensemble sera correct.

Pour tester régulièrement — très régulièrement — notre programme nous avons besoin d'outils qui vont automatiser ces tests mais commençons par les planifier.

## 9.2 Planifier les tests

Dans la méthode de résolution de problèmes que nous avons présenté dans le chapitre 1 p. 9 et dans l'exercice résolu de la section 4.1 p. 31 nous précisions que l'écriture d'une solution complète dans ce premier cours de développement consistait en :

1. spécifier le problème ;
2. fournir des exemples ;

---

53. Je m'autorise un accord de proximité.

54. Voir [https://fr.wikipedia.org/wiki/Test\\_\(informatique\)](https://fr.wikipedia.org/wiki/Test_(informatique))

3. écrire un algorithme ;
4. vérifier les exemples (en traçant l'algorithme) ;
5. écrire un programme exécutable correspondant à l'algorithme ;
6. tester le programme et constater qu'il fournit bien les résultats attendus.

Les points 2 et 4 sont les prémisses à l'écriture des tests puisque ces exemples vont nous convaincre que l'algorithme — et le programme — fournit bien les « bons » résultats. Ces exemples constituent le **plan de tests** que nous allons utiliser. Dans ces différents cas, outre les cas généraux, les cas particuliers, les valeurs limites, apparaîtront aussi des cas montrant les erreurs de programmation fréquentes. Par exemple :

- ▷ commencer ou arrêter trop tôt ou trop tard une boucle ;
- ▷ ne pas initialiser ou mal initialiser une variable ;
- ▷ confondre  $<$  et  $\leq$  ou  $>$  et  $\geq \dots$  voire  $<$  et  $>$  ;
- ▷ confondre AND et OR ;
- ▷ mal écrire la négation d'une proposition ;
- ▷ ...

C'est l'expérience qui permettra d'écrire un plan de tests efficace et convaincant.

Un **plan de test** est donc une liste ou un tableau reprenant un certain nombre d'exemples qui mettent en œuvre la méthode testée sur des cas tels que ceux décrits ci-dessus et permettent de faire apparaître des problèmes lors de l'exécution automatique. Chacun des tests sera défini par une méthode qui porte un nom qui explique l'objectif du test sans la nécessité de commentaires additionnels. Autant que possible, ce nom décrit aussi la portée ou le contexte de ce qui est vérifié.



**Remarque** Dans certaines méthodologies, la mise en exergue de l'importance des tests se fait en écrivant d'abord les tests avant le code. Voir par exemple Wikipedia<sup>55</sup> ou cette article d'*extreme programming*<sup>56</sup> [Wel00].

Dès lors que le plan de tests est écrit, nous pouvons nous intéresser aux outils qui vont permettre d'automatiser l'exécution des tests et faciliter leur écriture. En effet pour que nos tests soient utiles il faudra tester souvent — idéalement dès qu'une méthode est écrite — et « tout ». Ce sont toutes les méthodes qui sont testées parce que l'écriture d'une méthode pourrait amener une erreur dans une autre méthode. Dans ces conditions, pour que le développeur ou la développeuse teste son programme, ce doit être automatique.

Cette automatisation est fournie par JUnit.

## 9.3 JUnit

Cette section ne fait pas partie de la matière évaluée dans la partie théorique.

55. [https://fr.wikipedia.org/wiki/Test\\_driven\\_development](https://fr.wikipedia.org/wiki/Test_driven_development)

56. <http://www.extremeprogramming.org/rules/testfirst.html>

**Remarque préalable** JUnit n'est pas fourni avec le JDK (*java development kit*), c'est un *framework* de tests indépendant que l'on trouve sur [junit.org](http://junit.org). Il est donc a priori nécessaire de l'installer. Cependant il est installé automatiquement par les IDE tels que Netbeans et IntelliJ./



**JUnit** est un *framework* simple pour l'écriture de tests répétables.

Pour tester la méthode — `foo` — dans la classe `MyClass`, il faut écrire une classe `MyClassTest` qui contiendra les méthodes de tests. Le nom de la classe de test porte le nom de la classe qu'elle teste, suffixé par `Test`. Ensuite, il faut demander à JUnit — dès lors qu'il est installé — d'exécuter cette classe. Un clic dans un IDE ou une commande du style fait l'affaire :

```
$  
java org.junit.runner.JUnitCore my.package.MyClassTest
```

terminal

La classe `MyClassTest` contient plusieurs méthodes. Une méthode de test par cas. Une méthode de test :

- ▷ est autonome. Elle ne reçoit pas de paramètre et ne retourne rien ;
- ▷ contient des affirmations qui seront évaluées ;

*La méthode doit me retourner « vrai ».* : `assertTrue()`

*La méthode doit me retourner « faux ».* : `assertFalse()`

*La méthode doit me donner cette valeur si ses paramètres sont ceux-ci.* :

`assertEquals(<valeur attendue>, <valeur>)`

Ces trois méthodes `assertTrue`, `assertFalse`, `assertEquals` sont définies dans la classe `org.junit.Assert`. On peut les utiliser directement après les avoir préalablement importées :

```
1 import static org.junit.Assert.*;
```

java

- ▷ est précédée de l'annotation `@Test` permettant de la faire reconnaître comme un test unitaire ;
- ▷ n'a pas de mot clé `static`.

Un test aura donc l'allure suivante :

```
@Test  
2 public void max2_cas1(){  
    assertEquals(2, max2(-1, 2));  
}
```

java

Après lancement des tests, JUnit fournira un rapport l'allure suivante :

```
$  
JUnit version 4.12  
.  
  
Time: 0,012  
  
OK (1 test)
```

terminal

## 9.4 Exemple

Reprenons l'exemple du calcul du maximum de 2 nombres décrit dans la fiche 4 p 236. Voici un plan de tests :

test n°	description	nb1	nb2	réponse attendue
1	maximum en position 2 et nombre négatif	-3	4	4
2	maximum en position 1	7	4	7
3	tous égaux	4	4	4
4	avec un nul	0	-4	0

Et la classe de tests `MaximumTest` sans les commentaires.

```
package be.he2b.esi.cours.dev1;
2
import org.junit.Test;
import static org.junit.Assert.*;
5
public class MaximumTest {

8
    @Test
    public void max2_maximumEnPosition2EtUnNégatif(){
11        assertEquals(4, max2(-3, 4));
    }

14    @Test
    public void max2_maximumEnPosition1(){
        assertEquals(7, max2(7, 4));
17    }

    @Test
20    public void max2_tousÉgaux(){
        assertEquals(4, max2(4, 4));
    }

23    @Test
    public void max2_avecUnNul(){
26        assertEquals(0, max2(0, -4));
    }
}
```

java

```
$
JUnit version 4.12
....

Time: 0,012

OK (4 tests)
```

terminal

## Troisième partie

### Les tableaux





# Chapitre 10

## Les tableaux

Dans ce chapitre nous étudions les tableaux, une structure qui peut contenir plusieurs exemplaires de même type.

### Contenu

---

10.1	Utilité des tableaux . . . . .	<b>114</b>
10.2	Définitions . . . . .	<b>117</b>
10.3	Déclaration - création - initialisation . . . . .	<b>117</b>
10.3.1	Déclaration . . . . .	118
10.3.2	Création . . . . .	118
10.3.3	Initialisation . . . . .	118
10.4	Tableau et paramètres . . . . .	<b>120</b>
10.4.1	Un tableau comme paramètre en entrée . . . . .	120
10.4.2	Un tableau comme paramètre en entrée-sortie . . . . .	120
10.4.3	Un tableau comme paramètre en langage Java . . . . .	120
10.4.4	Un tableau comme type de retour . . . . .	121
10.4.5	Paramétrer la taille . . . . .	121
10.5	Parcours d'un tableau . . . . .	<b>122</b>
10.6	Taille logique et taille physique . . . . .	<b>123</b>
10.7	Les tableaux ont comme premier indice 0 . . . . .	<b>124</b>

---

## 10.1 Utilité des tableaux

Nous allons introduire la notion de tableau à partir d'un exemple dans lequel l'utilité de cette structure de données apparaîtra de façon naturelle.

**Exemple.** Statistiques de ventes.

Un gérant d'une entreprise commerciale souhaite connaître l'impact d'une journée de promotion publicitaire sur la vente de dix de ses produits.

Pour ce faire, les numéros de ces produits (numérotés de 0 à 9 pour simplifier) ainsi que les quantités vendues pendant cette journée de promotion sont encodés au fur et à mesure de leurs ventes. En fin de journée, le vendeur entrera la valeur -1 pour signaler la fin de l'introduction des données. Ensuite, les statistiques des ventes seront affichées.

La démarche générale se décompose en trois parties :

- ▷ le traitement de début de journée, qui consiste essentiellement à mettre les compteurs des quantités vendues pour chaque produit à 0 ;
- ▷ le traitement itératif durant toute la journée : au fur et à mesure des ventes, il convient de les enregistrer, c'est-à-dire d'ajouter au compteur des ventes d'un produit la quantité vendue de ce produit ; ce traitement itératif s'interrompra lorsque la valeur -1 sera introduite ;
- ▷ le traitement final, consistant à communiquer les valeurs des compteurs pour chaque produit.

Vous trouverez sur la page suivante une version possible de cet algorithme sans utiliser les tableaux.

```
import java.util.Scanner;
2 /*
   Calcule et affiche la quantité vendue de 10 produits
   */
5 public static void statistics(){
    Scanner keyboard = new Scanner(System.in);
    int cpt0, cpt1, cpt2, cpt3, cpt4, cpt5, cpt6, cpt7, cpt8, cpt9;
8   int productId, quantity;
    cpt0 = 0;
    cpt1 = 0;
11   cpt2 = 0;
    cpt3 = 0;
    cpt4 = 0;
14   cpt5 = 0;
    cpt6 = 0;
    cpt7 = 0;
17   cpt8 = 0;
    cpt9 = 0;
    System.out.print("Introduire le numéro de produit:");
20   productId = keyboard.nextInt();
    while (productId != -1){
        System.out.print("Quantité:");
23        quantity = keyboard.nextInt();
        switch (productId){
            case 0:
26            cpt0 = cpt0 + quantity;
                break;
            case 1:
29            cpt1 = cpt1 + quantity;
                break;
            case 2:
32            cpt2 = cpt2 + quantity;
                break;
            case 3:
35            cpt3 = cpt3 + quantity;
                break;
            case 4:
38            cpt4 = cpt4 + quantity;
                break;
            case 5:
41            cpt5 = cpt5 + quantity;
                break;
            case 6:
44            cpt6 = cpt6 + quantity;
                break;
            case 7:
47            cpt7 = cpt7 + quantity;
                break;
            case 8:
50            cpt8 = cpt8 + quantity;
                break;
            case 9:
53            cpt9 = cpt9 + quantity;
                break;
        }
56        System.out.print("Introduire le numéro de produit:");
        productId = keyboard.nextInt();
    }
59 }
```

```

System.out.println("Quantité vendue de produit_0: " + cpt0);
3 System.out.println("Quantité vendue de produit_1: " + cpt1);
System.out.println("Quantité vendue de produit_2: " + cpt2);
System.out.println("Quantité vendue de produit_3: " + cpt3);
6 System.out.println("Quantité vendue de produit_4: " + cpt4);
System.out.println("Quantité vendue de produit_5: " + cpt5);
System.out.println("Quantité vendue de produit_6: " + cpt6);
9 System.out.println("Quantité vendue de produit_7: " + cpt7);
System.out.println("Quantité vendue de produit_8: " + cpt8);
System.out.println("Quantité vendue de produit_9: " + cpt9);
12 }

```

java

Que se passerait-il si le nombre de produits à traiter était de 20, voire 1000 ? Une simplification de l'écriture s'impose. La solution est apportée par un nouveau type de variables : les **variables indicées** ou **tableaux** (*arrays* en anglais).

Au lieu d'avoir à manier dix compteurs distincts (*cpt0*, *cpt1*, etc.), nous allons envisager une seule « grande » variable *cpt* compartimentée en dix « cases » ou « sous-variables » (appelées aussi les « éléments » du tableau). Elles se distingueront les unes des autres par un numéro (un « indice ») : *cpt0* deviendrait ainsi *cpt[0]*, *cpt1* deviendrait *cpt[1]*, et ainsi de suite jusqu'à *cpt9* qui deviendrait *cpt[9]*.

	<i>cpt[0]</i>	<i>cpt[1]</i>	<i>cpt[2]</i>	<i>cpt[3]</i>	<i>cpt[4]</i>	<i>cpt[5]</i>	<i>cpt[6]</i>	<i>cpt[7]</i>	<i>cpt[8]</i>	<i>cpt[9]</i>
<i>cpt</i>										

Un des intérêts de cette notation est la possibilité de faire apparaître une variable entre les crochets, par exemple *cpt[i]*, ce qui permet une grande économie de lignes de code.

Voici la version de notre solution avec tableau.

```

import java.util.Scanner;
2 public static void statisticsWithArray(){
    Scanner keyboard = new Scanner(System.in);
    int[] cpt = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
5    int i;
    int productId;
    int quantity;
8    System.out.print("Introduire le numéro de produit:");
    productId = keyboard.nextInt();
    while (productId != -1){
11        System.out.print("Quantité:");
        quantity = keyboard.nextInt();

14        cpt[productId] = cpt[productId] + quantity;

        System.out.print("Introduire le numéro de produit:");
17        productId = keyboard.nextInt();
    }

20    for(int i=0; i<10; i++){
        System.out.println(
            "Quantité vendue de produit " i + ": " + cpt[i]);
23    }
}

26

```

java

## 10.2 Définitions

Un **tableau** est une suite d'éléments de même type portant tous le même nom mais se distinguant les uns des autres par un indice.



L'**indice** est un entier donnant la position d'un élément dans la suite. Cet indice varie entre la position du premier élément et la position du dernier élément, ces positions correspondant aux bornes de l'indice. Notons qu'il n'y a pas de « trou » : tous les éléments existent entre le premier et le dernier indice.

La **taille** d'un tableau est le nombre de ses éléments. Attention, la taille d'un tableau ne peut pas être modifiée pendant son utilisation, elle est déterminée à la création du tableau.

Un tableau est de type **référence**.

## 10.3 Déclaration - création - initialisation

Afin de pouvoir utiliser un tableau, il sera nécessaire de distinguer :



- ▷ la **déclaration** qui consiste à signaler qu'une variable est un tableau. Dans un programme, cette déclaration a pour effet de réserver un emplacement mémoire qui contiendra une référence vers les cases du tableau ;
- ▷ la **création** du tableau consiste à réserver l'emplacement mémoire qui contiendra les différentes valeurs. Pour le créer, il faut connaître sa taille ;

- ▷ l'**initialisation** du tableau consiste à donner ses premières valeurs aux différents éléments du tableau.

Nous supposons dans la suite que `length` est le nombre d'éléments — la longueur — et `Type` est le type des éléments que l'on trouvera dans le tableau.

Tous les types sont permis mais tous les éléments sont du même type.

### 10.3.1 Déclaration

Pour **déclarer** un tableau (grammaire simplifiée) :

*ArrayDeclarationStatement:*  
*Type*[] *Identifieur*

```
1 int[] myArray;  
  String[] strings;
```

java

Seule la variable est déclarée. Aucun espace mémoire n'est réservé pour recevoir les éléments du tableau. Ceci sera fait lors de la création du tableau.

### 10.3.2 Création

Pour **créer** un tableau, préalablement déclaré, nous écrivons :

*ArrayCreationExpression:*  
`new` *Type*[*Expression*]

```
myArray = new int[10];  
strings = new String[3];  
3
```

java

L'emplacement mémoire nécessaire pour stocker la valeur de chaque élément du tableau est réservé.

### 10.3.3 Initialisation

Pour **initialiser** les éléments du tableau, nous pouvons le faire en accédant à chaque élément du tableau. Un par un.

L'accès aux éléments d'un tableau se fait grâce aux crochets `myArray[i]` pour nos algorithmes, en langage Java et dans beaucoup d'autres langages. Comme suit :

```
myArray[0] = 5;  
strings[1] = "Hello";  
3
```

java

La première case du tableau porte l'indice 0 la dernière l'indice `size-1`. C'est une erreur d'indiquer un indice qui ne correspond pas à une case du tableau (trop petit ou trop grand). Par exemple, si on déclare et crée le tableau `myArray` :

```
int[] myArray = new int[100];
```

java

c'est une erreur d'utiliser `myArray[-1]` ou `myArray[100]`.

Il est également possible d'initialiser un tableau en une seule fois, en utilisant un *array initializer*. Une notation entre accolades.

```
myArray = {2, 3, 7}
```

langage naturel

Voici la grammaire Java.

```
ArrayInitializer:  
    { [VariableInitializerList] }
```

```
VariableInitializerList:  
    Expression [ , Expression ]
```

```
1  int[] myArray;  
   myArray = new int[3];  
   myArray[0] = 2;  
4  myArray[1] = 3;  
   myArray[2] = 7;  
  
7  myArray = new int[] {-1, 3, 12, 5};  
  
   int[] myArray2 = {3, 14, 15, 92};  
10
```

java

#### ligne 7

Pour utiliser un *ArrayInitializer*, il faut créer et initialiser le tableau dans la même instruction. Dans ce cas, le langage interdit de renseigner le nombre d'éléments du tableau... le compilateur peut le déduire.

Cette instruction crée un nouveau tableau qui écrase l'autre.

#### ligne 9

Lorsque l'on initialise le tableau avec un *ArrayInitializer* lors de la déclaration, la partie création peut être omise.

**Remarque.** Chaque élément d'un tableau doit être manié avec la même précaution qu'une variable simple, c'est-à-dire qu'on ne peut utiliser un élément du tableau qui n'aurait pas été préalablement affecté ou initialisé.

## 10.4 Tableau et paramètres

Le type *tableau* étant un type à part entière, il est tout-à-fait éligible comme type pour les paramètres et la valeur de retour d'un algorithme. De manière générale là où pouvait apparaître un type (entier, pseudo-réel...) peut apparaître un tableau. Toujours. Voyons cela en détail.

Nous avons vu dans la section 7.3 p. 81 les différentes sortes de paramètres. Voyons ce qu'il en est pour chacun d'eux lorsque ce paramètre est un tableau.

### 10.4.1 Un tableau comme paramètre en entrée

↓ : indique que l'algorithme va consulter les valeurs du tableau reçu en paramètre. Les éléments doivent donc avoir été initialisés avant d'appeler l'algorithme. Exemple :

```
// Affiche les éléments d'un tableau
algorithm display(myArray↓ : array of 10 integers)
  for i from 0 to 9
    | print myArray[i]

// Utilisations possible
array of 10 integers anArray
anArray = {2,3,5,7,11,13,17,19,23,29}
display(anArray)
```

*pseudocode*

### 10.4.2 Un tableau comme paramètre en entrée-sortie

↓↑ : indique que l'algorithme va consulter/modifier les valeurs du tableau reçu en paramètre. Exemple :

```
// Inverse le signe des éléments du tableau
algorithm reverseSign(myArray↓↑ : array of 10 integers)
  for i from 0 to 9
    | myArray[i] = -myArray[i]

// Utilisation possible
array of integers anArray
anArray = array of integers {2,-3,5,-7,11,13,17,-19,23,29}
reverseSign(anArray)
```

*pseudocode*

### 10.4.3 Un tableau comme paramètre en langage Java

En langage Java, le passage de paramètres se fait toujours par valeur. Comme un tableau est de type référence, c'est bien la valeur de la référence qui est passée en paramètre. En ce sens, il sera possible de modifier les valeurs d'un tableau reçu mais pas de remplacer le tableau reçu par un autre. Exemples :



```

1  public static void reverseSign(int[] is){
    for (int i=0; i < is.length; i = i + 1){
        is[i] = -is[i];
4  }
    }

7  // Utilisation
    int[] myArray = {2,-3,5,-7,11,13,17,-19,23,29};
    reverseSign(myArray);

```

java

```

    public static void fail(double[] ds){
        ds = new double[] {2, 2.61};
3  }

6  // Utilisation
    double[] myArray = {1, 3.14};
    fail(myArray);
9  // 3.14 et pas 2.61
    System.out.println("Value:␣" + myArray[1]);

```

java



Notez comme on utilise le nom de variable *i* pour *un entier* (dans la boucle), et les noms de variable *is* (pour *des entiers*, c'est-à-dire *un tableau d'entiers*) et *ds* (pour *des doubles*) pour les tableaux. Ajouter un *s* est une convention classique pour nommer un tableau. Si chaque élément d'un tableau est un *truc*, on peut appeler le tableau *trucs*.

#### 10.4.4 Un tableau comme type de retour

Comme pour n'importe quel autre type, un algorithme ou un programme peut retourner un tableau. Ce sera à lui de le déclarer et de lui donner des valeurs. Exemple :

```

    public static int[] create(){
2      int[] myArray = new int[10];
        for (int i=0; i < 10; i = i + 1){
            myArray[i] = i;
5      }
        return myArray;
    }

8

    // Utilisation possible
11   int[] anArray = create();

```

java

#### 10.4.5 Paramétrer la taille

Un tableau connaît sa taille dès lors qu'elle existe c'est-à-dire dès que le tableau est créé. La taille est accessible grâce au mot `length` comme suit :

```
1    double[] myArray;  
    int size = myArray.length;  
  
4    double[] otherArray = {3.14, 2.71};  
    int otherSize = otherArray.length;
```

java

Un tableau peut être passé en paramètre à un algorithme ou à une méthode et ce, quelle que soit la taille du tableau passé en paramètre. L'algorithme ou la méthode aura comme paramètre un tableau qui connaît sa taille. Lorsque l'algorithme ou la méthode recevra le paramètre effectif, la taille sera déterminée.

**Exemple :**

```
public static void display(int[] myArray){  
    System.out.println("Tableau de "  
3    + myArray.length  
    + " éléments." );  
    for (int i=0; i< myArray.length; i++){  
6        System.out.println(myArray[i]);  
    }  
  
9    // Utilisations possibles  
    int[] integers = new int[2];  
    integers[0] = 56;  
12   integers[1] = -3;  
    display(integers);  
  
15   int[] myBeautifulArray = {6, 12, 18};  
    display(myBeautifulArray);  
}  
18
```

java

## 10.5 Parcours d'un tableau

Dans la plupart des problèmes que vous rencontrerez vous serez amenés à parcourir un tableau. Il est important de maîtriser ce parcours. Examinons les situations courantes et voyons quelles solutions conviennent.

### Parcours complet

Nous avons déjà vu dans les exemples comment parcourir tous les éléments d'un tableau. Une boucle for fait généralement l'affaire comme ceci :

```
int[] myArray = { /* insert values */ };  
for (int index = 0; index < myArray.length; index++){  
3    // do something with myArray[index]  
}
```

java

Java propose une autre forme de `for` — l'*enhanced for* ou *foreach* — permettant de parcourir un tableau<sup>57</sup>

```
1 int[] myArray = { /* insert values */ };  
  for (int value: myArray){  
    // do something with value  
4 }
```



La fiche 5 page 239 décrit comment afficher tous les éléments d'un tableau. Il est possible de faire autre chose avec ces éléments : les sommer, les comparer. . .

### Parcours partiel

Parfois, il n'est pas nécessaire de parcourir le tableau jusqu'au bout car un arrêt prématuré survient dès qu'une condition est remplie.

Par exemple : recherche d'un élément et arrêt dès qu'il est trouvé ; vérification que toutes les valeurs sont non nulles mais l'on vient d'en trouver une ; vérification que la suite des éléments est positives alors que l'on vient d'en trouver un négatif ou nul.

Pour résoudre ce problème, la structure **tant que** avec un test d'arrêt fait bien l'affaire.

La fiche 7 page 244 détaille un parcours partiel.

## 10.6 Taille logique et taille physique

Parfois, la taille du tableau n'est pas connue à l'avance.

Imaginons, par exemple, qu'il s'agisse de demander des valeurs à l'utilisateur et de les stocker dans un tableau pour un traitement ultérieur. Supposons que l'utilisateur va indiquer la fin des données par une valeur sentinelle. Impossible de savoir, *a priori*, combien de valeurs il va entrer et, par conséquent, la taille à donner au tableau.

Une solution est de créer un tableau suffisamment grand pour tous les cas<sup>58</sup> quitte à n'en n'utiliser qu'une partie.

Mais comment savoir quelle est la partie du tableau qui est effectivement utilisée ?

Comprenez bien qu'il **n'y a pas de concept de case vide**. Rien ne différencie une case utilisée d'une case non utilisée. La manière de s'en sortir est de stocker les valeurs dans la partie basse du tableau (à gauche) et de retenir dans une variable le nombre de cases effectivement utilisées.

La **taille physique** d'un tableau est le nombre de cases qu'il contient. Sa **taille logique** est le nombre de cases actuellement utilisées.



**Exemple.** Voici un tableau qui ne contient pour l'instant que quatre nombres.

<sup>57</sup>. Nous verrons en DEV2, que cette structure permet de parcourir d'autres choses que les tableaux.

<sup>58</sup>. En tout cas, suffisamment grand pour tous les cas qu'on accepte de prendre en compte ; il faudra bien fixer une limite.

10	4	3	7	?	?	?	?	?	?
----	---	---	---	---	---	---	---	---	---

taillePhysique = 10      tailleLogique = 4

Les cases indiquées par "?" ne sont pas vides ; elles peuvent être non initialisées (et il n'est pas possible de tester si une case est non initialisée) ou bien contenir une valeur qui n'est pas pertinente.

**Exemple.** L'algorithme suivant demande des valeurs positives à l'utilisateur et les stocke dans un tableau (maximum 1000). Toute valeur négative ou nulle est une valeur sentinelle.

```

1 public static void stockValues(){
    Scanner keyboard = new Scanner(System.in);
    int[] myArray = new int[1000];
4   int nbValues = 0;
    int value;
    value = keyboard.nextInt();
7   while (value > 0 && nbValues < myArray.length){
        myArray[nbValues] = value;
        nbValues++;
10    value = keyboard.nextInt();
    }
    // Test de value et pas de nbValues. Pourquoi ?
13   if (value > 0){
        System.out.println("La limite physique a été atteinte");
    }
16 }

```

java

En langage Java, essayer d'ajouter un élément en dehors de la taille du tableau va générer une erreur à l'exécution. Une `ArrayIndexOutOfBoundsException`.

## 10.7 Les tableaux ont comme premier indice 0

Les tableaux ont comme premier indice 0, Un tableau de taille  $n$  a toujours ses indices allant de 0 à  $n-1$ .

Parfois le problème à résoudre donne envie d'avoir des indices dont le premier ne commence pas à 0. Par exemple allant de 1 à 7. Dans ces cas, il est toujours possible de s'arranger pour commencer à partir de 0.

**Exemple.** Imaginons l'algorithme qui permet de convertir un numéro de jour en son intitulé : 1 donne "lundi", 2 donne "mardi", ... Une solution possible, sans tableau, serait :

```

1 public static String dayName(int dayNumber){
    String dayName;
    switch (dayNumber){
4      case 1:
        dayName = "lundi";
        break;
7      case 2:
        dayName = "mardi";
        break;
10     case 3:
        dayName = "mercredi";
        break;
13     case 4:
        dayName = "jeudi";
        break;
16     case 5:
        dayName = "vendredi";
        break;
19     case 6:
        dayName = "samedi";
        break;
22     case 7:
        dayName = "dimanche";
        break;
25     default:
        dayName = "inconnu";
    }
28    return dayName;
}

```

java

Mais une solution plus élégante passe par l'utilisation d'un tableau. Appelons-le `dayString` et stockons-y les noms des jours comme illustré ci-dessous.

1	2	3	4	5	6	7
"lundi"	"mardi"	"mercredi"	"jeudi"	"vendredi"	"samedi"	"dimanche"

Si le tableau commence à l'indice 1 jusqu'à l'indice 7, pour obtenir le nom d'un jour, il suffirait d'écrire : `dayString[dayNumber]`.

Comme les tableaux commencent toujours à l'indice 0, il faudra plutôt considérer le tableau ci-dessous et écrire `dayString[dayNumber-1]`.

0	1	2	3	4	5	6
"lundi"	"mardi"	"mercredi"	"jeudi"	"vendredi"	"samedi"	"dimanche"



# Chapitre 11

## Algorithmes d'insertions et de recherches

Souvent lorsque les applications stockent des données. Ces données changent. Il est nécessaire d'en ajouter, de modifier certaines, d'en rechercher, d'en supprimer.

### Contenu

11.1	Insertion dans un tableau non trié . . . . .	<b>128</b>
11.1.1	Inscription . . . . .	128
11.1.2	Vérifier une inscription . . . . .	129
11.1.3	Désinscription . . . . .	130
11.1.4	Exercices . . . . .	130
11.2	Insertion dans un tableau trié . . . . .	<b>131</b>
11.2.1	Rechercher la position d'une inscription . . . . .	131
11.2.2	Inscrire un étudiant . . . . .	133
11.2.3	Désinscription . . . . .	134
11.2.4	Intérêt de trier les valeurs ? . . . . .	135
11.3	La recherche dichotomique . . . . .	<b>135</b>

**Exemple** Imaginons qu'une école organise un événement libre et gratuit pour les étudiants et les étudiantes. Mais pour y assister, il est nécessaire de s'inscrire. Nous voulons fournir ce qu'il faut pour :

- ▷ inscrire un ou une étudiant · e ;
- ▷ vérifier l'inscription ;
- ▷ désinscrire ;
- ▷ afficher la liste des personnes inscrites.

Nous allons, pour l'exemple, stocker les numéros d'étudiants et d'étudiantes dans un tableau<sup>59</sup>. Comme le tableau ne sera généralement pas rempli, nous allons créer un tableau « trop grand » et gérer sa taille logique. Nous avons deux variables :

- ▷ `registered` : le tableau des numéros d'étudiants
- ▷ `nRegistered` : le nombre d'étudiant · es actuellement inscrit · es (la taille logique du tableau)

Nous allons envisager deux cas :

1. les numéros seront placés dans le tableau sans ordre imposé ;
2. les numéros seront placés dans l'ordre.

## 11.1 Insertion dans un tableau non trié

Les valeurs sont stockées dans le tableau dans un ordre quelconque. Par exemple :

<code>registered =</code>								
42010	41390	43342	42424	?	?	?	?	...
<code>nRegistered = 4</code>								

Il y a pour le moment quatre personnes inscrites dont les numéros sont ceux repris dans les quatre premières cases du tableau.

### 11.1.1 Inscription

Pour inscrire un étudiant ou une étudiante, il suffit de l'ajouter à la suite dans le tableau. Par exemple, en partant de la situation décrite ci-dessus, l'inscription de l'étudiant numéro 42123 aboutit à la situation suivante :

<code>registered =</code>								
42010	41390	43342	42424	42123	?	?	?	...
<code>nRegistered = 5</code>								

---

59. Ce n'est bien sûr pas la bonne manière de faire pour une application réelle pour laquelle nous utiliserions probablement une base de données... mais c'est une autre histoire... et un autre cours.



L'algorithme est assez simple :

```
algorithme register(registered, nRegistered, number)
    registered[nRegistered] = number
    incrémenter nRegistered
```

*langage naturel*

En langage Java, il n'est pas possible de passer `nRegistered` en entrée sortie puisque le passage de paramètre se fait par valeur et que `int` est un type primitif. Il faudra que ce soit en valeur de retour. Nous pourrions écrire :

```
public static int register(int[] registered, int nRegistered,
    int number){
3    // On peut imaginer vérifier que l'étudiant n'est pas déjà
    inscrit
    // On peut imaginer vérifier qu'il reste de la place
    // (c-à-d que le tableau n'est pas plein)
6    registered[nRegistered] = number;
    return nRegistered + 1;
9    }
9
```

java

Dans ce cas, c'est bien le code appelant qui devra maintenir la valeur de la taille logique à jour.

### 11.1.2 Vérifier une inscription

Pour vérifier si un étudiant est déjà inscrit, il faut le rechercher dans la partie utilisée du tableau. Cette recherche se fait via un parcours avec sortie prématurée.

L'algorithme ne va pas se contenter de retourner un booléen précisant si le numéro est trouvé mais va retourner l'indice dans le tableau de ce numéro, -1 sinon.

```
public static int check(
    int[] registered,
3    int nRegistered,
    int number){
    int i = 0;
6    while (i < nRegistered && registered[i] != number){
        i++;
    }
9    if (i < nRegistered){
        return i;
    } else {
12    return -1;
    }
15    }
15
```

java

La fiche 8 page 247 reprend cet algorithme dans un cadre plus général.

### 11.1.3 Désinscription

Pour désinscrire un étudiant, il faut le trouver dans le tableau et l'enlever. Attention, il ne peut pas y avoir de trou dans un tableau (il n'y a pas de case vide). Le plus simple est d'y déplacer le dernier élément. Par exemple, reprenons la situation après inscription de l'étudiant numéro 42123.

registered =								
42010	41390	43342	42424	42123	?	?	?	...
nRegistered = 5								

La désinscription de l'étudiant numéro 41390 donnerait ce qui suit.

Nous avons volontairement indiqué le 42123 en 5<sup>e</sup> position. Il est toujours là mais ne sera pas considéré par les algorithmes puisque cette case est au-delà de la taille logique (donnée par la variable `nRegistered`).

registered =								
42010	42123	43342	42424	42123	?	?	?	...
nRegistered = 4								

L'algorithme est assez simple à écrire en utilisant la recherche écrite juste avant :

```

public static int unsubscribe(int[] registered,
    int nRegistered, int number){
3   int pos = check(registered, nRegistered, number);
    registered[pos] = registered[nRegistered - 1];
    return nRegistered - 1;
6 }
```

java

### 11.1.4 Exercices

1. Éviter la double inscription.  
Comment modifier l'algorithme d'inscription pour s'assurer qu'un étudiant ne s'inscrive pas deux fois ?
2. Limiter le nombre d'inscriptions.  
Comment modifier l'algorithme d'inscription pour refuser une inscription si le nombre maximal de participant est atteint en supposant que ce maximum est égal à la taille physique du tableau ?
3. Vérifier la désinscription.  
Que se passerait-il avec l'algorithme de désinscription tel qu'il est si on demande à désinscrire un étudiant non inscrit ? Que suggérez-vous comme changement ?
4. Optimiser la désinscription.  
Dans l'algorithme de désinscription tel qu'il est, voyez-vous un cas où on effectue une opération inutile ? Avez-vous une proposition pour l'éviter ?

## 11.2 Insertion dans un tableau trié

Imaginons à présent que nous maintenons un ordre dans les données du tableau. En reprenant l'exemple utilisé pour les données non triées, nous avons :

registered =								
41390	42010	42424	43342	?	?	?	?	...
nRegistered = 4								

- Qu'est-ce que ça change au niveau des algorithmes ?
- Beaucoup !

Par exemple, plus question de placer un nouvel inscrit à la suite des autres ; il faut trouver sa place.

### 11.2.1 Rechercher la position d'une inscription

Tous les algorithmes que nous allons voir dans le cadre de données triées ont une partie en commun : la recherche de la position du numéro, s'il est présent ou de la position où il aurait dû être s'il est absent. Commençons par cela.

L'algorithme est assez proche de celui de la vérification d'une inscription vu précédemment si ce n'est que l'algorithme peut s'arrêter dès qu'un numéro plus grand est trouvé. Nous allons décomposer l'algorithme en deux parties : la première pour la recherche de la position à laquelle devrait se trouver le numéro et la seconde pour vérifier que c'est bien le bon numéro qui se trouve à cette position.

```

/*
2  * Recherche d'un étudiant ou d'une étudiante par son numéro d'id.
  * return: l aposition où a été trouvé le numéro
  * ou -1 si non présent
5  */
public static int find(
    int[] registered,
8   int nRegistered,
    int number){
    int pos = findPosition(registered, nRegistered, number);
11   if (registered[pos] == number){
        return pos;
    } else {
14     return -1;
    }
}

17
/*
  * Recherche d'une position
20 * return: la position à laquelle est ou devrais être le numéro
  */
public static int findPosition(
23   int[] registered,
    int nRegistered,
    int number){
26   int pos = 0,
    while (pos < nRegistered && registered[pos] < number){
        pos++;
29   }
    return pos;
}

32

```

java

Comprenez-vous pourquoi :

- ▷ `registered[nRegistered] < number` contient un '`<`' dans la condition du tant que et pas un '`≠`' ?
- ▷ la condition du `if` est composée de deux parties et utilise un `=` ?

La liste étant triée il faut s'arrêter de chercher dès que la valeur n'est plus inférieure à la valeur cherchée. Dès qu'elle est plus grande c'est que la valeur **n'est pas** dans la liste.

De même pour la condition du `if`, il faut d'abord tester si la recherche n'est pas arrivée à la fin du nombre d'inscrits et d'inscrites. Ensuite, il faut voir si le numéro à la position trouvée est bien le bon.

Le `if` tel que présenté dans l'algorithme peut se réécrire grâce à l'opérateur ternaire de Java. Profitons en pour le présenter.

*ConditionalExpression:*

*Condition ? Expression : Expression*

Si la condition est vraie, l'expression prend la première valeur et si la condition est fausse, ce sera la seconde valeur. Par exemple : `heure < 12 ? "matin" : "soir"` vaut "soir" si `heure` vaut 15.

```

1  /**
   * Recherche une inscription dans le tableau.
   *
4  * @param registered le tableau d'inscrit·es
   * @param nRegistered le nombre d'inscrit·es
   * @param number le numéro à chercher
7  * @return la position dans le tableau, -1 si non présent
   */
   public static int find(int[] registered, int nRegistered,
10      int number){
       int pos = findPosition(registered, nRegistered, number);
       return registered[pos] == number ? pos : -1;
13  }

   /**
16  * Recherche la position que le nombre soit présent ou pas.
   * @param registered le tableau d'inscrit·es
   * @param nRegistered le nombre d'inscrit·es
19  * @param number le numéro à chercher
   * @return la position à laquelle est ou devrait être le numéro.
   */
22  public static int findPosition(int[] registered, int nRegistered,
       int number){
       int pos = 0;
25     while (pos < nRegistered && registered[pos] < number){
           pos = pos + 1;
       }
28     return pos;
   }

```

java

### 11.2.2 Inscrire un étudiant

L'inscription d'un étudiant se fait en trois étapes :

1. recherche de l'étudiant via l'algorithme vu à l'instant, ce qui nous donne la place où le placer ;  
Nous ne vérifions pas si l'étudiant ou l'étudiante est déjà présent·e dans le tableau.
2. libération de la place pour l'y mettre ;  
Cette opération n'est pas triviale. Si vous tenez des cartes en main, il est facile d'insérer une nouvelle carte à n'importe quelle position. Avec un tableau, il en va tout autrement ; pour insérer un élément à un endroit donné, il faut décaler tous les suivants d'une position sur la droite.
3. placer l'élément à la position libérée.

Ce qui nous donne :

```

/**
 * Inscrit un ou une étudiant·e.
3  *
 * @param registered le tableau d'inscrit·es
 * @param nRegistered le nombre d'inscrit·es
6  * @param number le numéro à enregistrer
 */
public static int subscribe(int[] registred, int nRegistered,
9     int number){
    int pos = findPosition(registered, nRegistered, number);
    shiftRight(registred,pos, nRegistered-1);
12    registered[pos] = number;
    return nRegistered + 1;
}
15
/**
 * Décale les éléments du tableau de begin à end un à droite.
18 * @param tab le tableau
 * @param begin l'indice de départ
 * @param end l'indice de fin
21 */
public static void shiftRight(int[] tab, int begin, int end){
    for (int i = end; i >= begin; i = i - 1){
24        tab[i + 1] = tab[i];
    }
}
27

```

java

### 11.2.3 Désinscription

Ici, il va falloir décaler vers la gauche les éléments qui suivent celui à supprimer.

```

/**
 * Désinscrit un ou une étudiant·e.
3  *
 * @param registered le tableau d'inscrit·es
 * @param nRegistered le nombre d'inscrit·es
6  * @param number le numéro à supprimer
 */
public static int unsubscribe(int[] registred, int nRegistered,
9      int number){
    int pos = findPosition(registered, nRegistered, number);
    shiftLeft(registred,pos+1, nRegistered);
12  return nRegistered - 1;
}

15 /**
 * Décale les éléments du tableau de begin à end un à gauche
 * (et écrase donc l'élément en position begin-1).
18 * @param tab le tableau
 * @param begin l'indice de départ
 * @param end l'indice de fin
21 */
public static void shiftLeft(int[] tab, int begin, int end){
    for (int i = begin; i <= end; i = i + 1){
24        tab[i - 1] = tab[i];
    }
}
27

```

java

La fiche 9 page 251 reprend cet algorithme dans un cadre plus général.

#### 11.2.4 Intérêt de trier les valeurs ?

Est-ce que trier les valeurs est vraiment intéressant ?

La recherche est un peu plus rapide. En moyenne deux fois plus rapide si l'élément ne s'y trouve pas. Par contre, l'ajout et la suppression d'un élément sont plus lents. Les algorithmes sont plus complexes à écrire et à comprendre. Le gain ne paraît pas évident et en effet, en l'état, il ne l'est pas.

Mais c'est sans compter un autre algorithme de recherche, beaucoup plus rapide, la recherche dichotomique, que nous allons voir maintenant.

### 11.3 La recherche dichotomique

La recherche dichotomique est un algorithme de recherche d'une valeur dans un tableau trié. Il a pour essence de réduire à chaque étape la taille de l'ensemble de recherche de moitié, jusqu'à ce qu'il ne reste qu'un seul élément dont la valeur devrait être celle recherchée, sauf bien entendu en cas d'inexistence de cette valeur dans l'ensemble de départ.

Pour l'expliquer, nous allons considérer un tableau d'entiers complètement rempli. Il sera facile d'adapter la solution à un tableau partiellement rempli (avec une taille logique) ou un tableau contenant tout autre type de valeurs qui peut s'ordonner.

### Description de l'algorithme

Soit *value* la valeur recherchée dans une zone délimitée par les indices *leftIndex* et *rightIndex*. Il faut commencer par déterminer l'élément médian, c'est-à-dire celui qui se trouve « au milieu » de la zone de recherche<sup>60</sup> ; son indice sera déterminé par la formule

$$\text{medianIndex} = (\text{leftIndex} + \text{rightIndex}) \text{ DIV } 2$$



L'algorithme compare alors *value* avec la valeur de cet élément médian ; il est possible que ce soit la valeur cherchée ; sinon, il partage la zone de recherche en deux parties : une qui **ne contient certainement pas** la valeur cherchée et une qui **pourrait la contenir**. C'est cette deuxième partie qui devient la nouvelle zone de recherche. Il adapte *leftIndex* ou *rightIndex* en conséquence. L'algorithme réitère le processus jusqu'à ce que la valeur cherchée soit trouvée ou que la zone de recherche soit réduite à sa plus simple expression, c'est-à-dire un seul élément.

### Exemple de recherche fructueuse

Supposons que l'on recherche la valeur **23** dans un tableau de 20 entiers. La zone ombrée représente à chaque fois la partie de recherche, qui est au départ le tableau trié dans son entièreté. Au départ, *leftIndex* vaut 0 et *rightIndex* vaut 19.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

*Première étape* :  $\text{medianIndex} = (0 + 19) \text{ DIV } 2$ , c'est-à-dire 9.

Comme la valeur en position 9 est 15, la valeur cherchée doit se trouver à sa droite, et on prend comme nouvelle zone de recherche celle délimitée par *leftIndex* qui vaut 10 et *rightIndex* qui vaut 19.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

*Deuxième étape* :  $\text{medianIndex} = (10 + 19) \text{ DIV } 2$ , c'est-à-dire 14.

Comme on y trouve la valeur 25, on garde les éléments situés à la gauche de celui-ci ; la nouvelle zone de recherche est [10, 13].

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

*Troisième étape* :  $\text{medianIndex} = (10 + 13) \text{ DIV } 2$ , c'est-à-dire 11 où se trouve l'élément 20. La zone de recherche devient *leftIndex* vaut 12 et *rightIndex* vaut 13.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

60. Cet élément médian n'est pas tout à fait au milieu dans le cas d'une zone contenant un nombre pair d'éléments.



Quatrième étape :  $\text{medianIndex} = (12 + 13) \text{ DIV } 2$ , c'est-à-dire 12 où se trouve la valeur cherchée ; la recherche est terminée.

### Exemple de recherche infructueuse

Recherchons à présent la valeur **8**. Les étapes de la recherche vont donner successivement

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29
1	3	5	7	7	9	9	10	10	15	16	20	23	23	25	28	28	28	29	29

Arrivé à ce stade, la zone de recherche s'est réduite à un seul élément. Ce n'est pas celui qu'on recherche mais c'est à cet endroit qu'il se serait trouvé ; c'est donc là qu'on pourra éventuellement l'insérer. Le paramètre de sortie prend la valeur 5.

### Algorithme

```

public static int binarySearch(int[] myArray, int value){
    int leftIndex = 0;
3   int medianIndex = -1;
    int rightIndex = myArray.length - 1;
    int candidate;
6   boolean isFound = false;

    while (!isFound && leftIndex <= rightIndex){
9       medianIndex = (leftIndex + rightIndex) / 2;
        candidate = myArray[medianIndex];
        if (candidate == value){
12            isFound = true;
        } else if (value < candidate){
            rightIndex = medianIndex - 1;
15        } else {
            leftIndex = medianIndex + 1;
        }
18    }

    if (isFound){
21        return medianIndex;
    } else {
        return -1;
24    }
}

```

java



# Chapitre 12

## Algorithmes de tris

Dans ce chapitre nous voyons quelques algorithmes simples pour trier un ensemble d'informations : recherche des maxima, tri par insertion et tri bulle dans un tableau. Des algorithmes plus efficaces seront vus en deuxième année.

### Contenu

12.1	Motivation . . . . .	<b>139</b>
12.2	Tri par insertion . . . . .	<b>141</b>
12.3	Tri par sélection des minima successifs . . . . .	<b>143</b>
12.4	Tri bulle . . . . .	<b>145</b>
12.5	Cas particuliers . . . . .	<b>146</b>
12.6	Références . . . . .	<b>147</b>

### 12.1 Motivation

Dans le chapitre précédent, nous avons vu que la recherche d'information est beaucoup plus rapide dans un tableau trié grâce à l'algorithme de recherche dichotomique. Nous avons vu comment **garder** un ensemble trié en insérant toute nouvelle valeur au *bon* endroit.

Dans certaines situations, nous serons amené à devoir trier toute une collection de valeurs. Par exemple un tableau.

1. D'abord les situations impliquant le classement total d'un ensemble de données « brutes », c'est-à-dire complètement désordonnées.

Prenons pour exemple les feuilles récoltées en vrac à l'issue d'un examen ; il y a peu de chances que celles-ci soient remises à l'examineur de manière ordonnée ; celui-ci devra donc procéder au tri de l'ensemble des copies, par exemple par ordre alphabétique des noms des étudiants ou des étudiantes, ou par numéro de groupe, etc.

2. Enfin, les situations qui consistent à devoir retrier des données préalablement ordonnées sur un autre critère.

Prenons l'exemple d'un paquet de copies d'examen déjà triées sur l'ordre alphabétique des noms des étudiants et des étudiantes, et qu'il faut retrier cette

fois-ci sur les numéros de groupe. Il est clair qu'une méthode efficace veillera à conserver l'ordre alphabétique déjà présent dans la première situation afin que les copies apparaissent dans cet ordre dans chacun des groupes.

Le dernier cas illustre un classement sur une **clé complexe** (ou **composée**) impliquant la comparaison de plusieurs champs d'une même structure : le premier classement se fait sur le numéro de groupe, et à numéro de groupe égal, l'ordre se départage sur le nom de la personne. Nous dirons de cet ensemble qu'il est classé en **majeur** sur le numéro de groupe et en **mineur** sur le nom d'étudiant.

Notons que certains tris sont dits **stables** parce qu'en cas de tri sur une nouvelle clé, l'ordre de la clé précédente est préservé pour des valeurs identiques de la nouvelle clé, ce qui évite de faire des comparaisons sur les deux champs à la fois. Les méthodes nommées **tri par insertion**, **tri bulle** et **tri par recherche de minima successifs** (que nous allons aborder dans ce chapitre) sont stables.

Certains tris sont évidemment plus performants que d'autres. Le choix d'un tri particulier dépend de la taille de l'ensemble à trier et de la manière dont il se présente, c'est-à-dire déjà plus ou moins ordonné. La performance quant à elle se juge sur deux critères :

- ▷ le nombre de tests effectués (comparaisons de valeurs) et ;
- ▷ le nombre de transferts de valeurs réalisés.

Les algorithmes classiques de tri présentés dans ce chapitre le sont surtout à titre pédagogique. Ils ont tous une « complexité en  $n^2$  » (on dit qu'ils sont « en  $\mathcal{O}(n^2)$  »), ce qui veut dire que si  $n$  est le nombre d'éléments à trier, le nombre d'opérations élémentaires (tests et transferts de valeurs) est proportionnel à  $n^2$ . Ils conviennent donc pour des ensembles de taille « raisonnable », mais peuvent devenir extrêmement lents à l'exécution pour le tri de grands ensembles, comme par exemple les données de l'annuaire téléphonique. Plusieurs solutions existent, comme la méthode de tri **Quicksort**. Cet algorithme très efficace faisant appel à la récursivité et qui sera étudié en deuxième année a une complexité en  $\mathcal{O}(n \log(n))$  en général et dans de rares cas en  $\mathcal{O}(n^2)$ .



Dans ce chapitre, les algorithmes traiteront du tri dans un **tableau d'entiers à une dimension**. Toute autre situation peut bien entendu se ramener à celle-ci moyennant la définition de la relation d'ordre propre au type de données utilisé. Ce sera par exemple l'ordre alphabétique pour les chaînes de caractères, l'ordre chronologique pour des objets **Date** ou **Moment** (que nous verrons plus tard), etc. De plus, le seul ordre envisagé sera l'ordre **croissant** des données. Plus loin, nous envisagerons le tri d'autres structures de données.

Enfin, dans toutes les méthodes de tri abordées, nous supposons la taille physique du tableau à trier égale à sa taille logique, celle-ci n'étant pas modifiée par l'action de tri. Nous supposons trier tout le tableau.

## 12.2 Tri par insertion

Cette méthode de tri repose sur le principe d'insertion de valeurs dans un tableau ordonné.

**Description de l'algorithme.** Le tableau à trier sera à chaque étape subdivisé en deux sous-tableaux : le premier cadré à gauche contiendra des éléments déjà ordonnés, et le second, cadré à droite, ceux qu'il reste à insérer dans le sous-tableau trié. Celui-ci verra sa taille s'accroître au fur et à mesure des insertions, tandis que celle du sous-tableau des éléments non triés diminuera progressivement.

Au départ de l'algorithme, le sous-tableau trié est le premier élément du tableau. Comme il ne possède qu'un seul élément, ce sous-tableau est donc bien ordonné ! Chaque étape consiste ensuite à prendre le premier élément du sous-tableau non trié et à l'insérer à la bonne place dans le sous-tableau trié.

Prenons comme exemple un tableau `tab` de 20 entiers. Au départ, le sous-tableau trié est formé du premier élément, `tab[0]`, qui vaut 20 :

<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>10</i>	<i>11</i>	<i>12</i>	<i>13</i>	<i>14</i>	<i>15</i>	<i>16</i>	<i>17</i>	<i>18</i>	<i>19</i>
20	12	18	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

L'étape suivante consiste à insérer `tab[1]`, qui vaut 12, dans ce sous-tableau de taille 2 :

<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>10</i>	<i>11</i>	<i>12</i>	<i>13</i>	<i>14</i>	<i>15</i>	<i>16</i>	<i>17</i>	<i>18</i>	<i>19</i>
12	20	18	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

Ensuite, c'est au tour de `tab[2]`, qui vaut 18, d'être inséré :

<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>10</i>	<i>11</i>	<i>12</i>	<i>13</i>	<i>14</i>	<i>15</i>	<i>16</i>	<i>17</i>	<i>18</i>	<i>19</i>
12	18	20	17	15	14	15	16	18	17	12	14	16	18	15	15	19	11	11	13

Et ainsi de suite jusqu'à insertion du dernier élément dans le sous-tableau trié.

**Algorithme.** L'algorithme combine la recherche de la position d'insertion et le décalage concomitant de valeurs.

```
1 /**
   * Tri par insertion.
   *
4  * @param myArray tableau à trier
   */
   public static void insertionSort(int[] myArray) {
7       for (int nbSortedValues = 1;
           nbSortedValues < myArray.length;
           nbSortedValues++) {
10          int value = myArray[nbSortedValues];
           int pos = nbSortedValues - 1;
           while (pos >= 0 && value < myArray[pos]) {
13              myArray[pos + 1] = myArray[pos];
              pos = pos - 1;
           }
16          myArray[pos + 1] = value;
       }
   }
```

java

## 12.3 Tri par sélection des minima successifs

Dans ce tri, l'algorithme recherche à chaque étape la plus petite valeur de l'ensemble non encore trié et la place immédiatement à sa position définitive.

**Description de l'algorithme.** Prenons par exemple un tableau de 20 entiers.

La première étape consiste à rechercher la valeur minimale du tableau. Il s'agit de l'élément d'indice 9 et de valeur 17.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
20	52	61	47	82	64	95	66	84	17	32	24	46	48	75	55	19	61	21	30

Celui-ci devrait donc apparaître en 1<sup>re</sup> position du tableau. Or cette position est occupée par la valeur 20. Comment procéder dès lors pour ne perdre aucune valeur et sans faire appel à un second tableau ? La solution est simple, il suffit d'échanger le contenu des deux éléments d'indices 0 et 9 :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
17	52	61	47	82	64	95	66	84	20	32	24	46	48	75	55	19	61	21	30

Le tableau se subdivise à présent en deux sous-tableaux, un sous-tableau déjà trié (pour le moment réduit au seul élément  $tab[0]$ ) et le sous-tableau des autres valeurs non encore triées (de l'indice 1 à 19). On recommence ce processus dans ce second sous-tableau : le minimum est à présent l'élément d'indice 16 et de valeur 19. Celui-ci viendra donc en 2<sup>e</sup> position, échangeant sa place avec la valeur 52 qui s'y trouvait :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
17	19	61	47	82	64	95	66	84	20	32	24	46	48	75	55	52	61	21	30

Le sous-tableau trié est maintenant formé des deux premiers éléments, et le sous-tableau non trié par les 18 éléments suivants.

Et ainsi de suite, jusqu'à obtenir un sous-tableau trié comprenant tout le tableau. En pratique l'arrêt se fait une étape avant car lorsque le sous-tableau non trié n'est plus que de taille 1, il contient nécessairement le maximum de l'ensemble.

```
/**
 * Trie le tableau reçu en paramètre via un tri par sélection
3  * des minima successifs
 *
 * @param myArray le tableau à trier
6 */
public static void selectionSort(int[] myArray) {
    for (int i = 0; i < myArray.length - 1; i++) {
9        int minIndex = searchMinIndex(myArray, i, myArray.length -
            1);
        if (minIndex != i) {
            swap(myArray, i, minIndex);
12        }
    }
}

15 /**
 * Retourne l'indice du minimum entre les indices début et fin
18 * du tableau reçu.
 *
 * En cas d'égalité, c'est l'indice le plus petit qui est retourné.
21 *
 * @param myArray le tableau d'entiers
 * @param begin l'indice de début
24 * @param end l'indice de fin
 */
public static int searchMinIndex(
27     int[] myArray,
        int begin,
        int end) {
30     int minIndex = begin;
    for (int i = begin + 1; i <= end; i++) {
        if (myArray[i] < myArray[minIndex]) {
33             minIndex = i;
        }
    }
36     return minIndex;
}

39 /**
 * Échange deux valeurs dans un tableau
 *
 * @param myArray un tableau
 * @param pos1 une position
 * @param pos2 une (autre) position
45 */
private static void swap(int[] myArray, int pos1, int pos2) {
    int value = myArray[pos1];
48     myArray[pos1] = myArray[pos2];
    myArray[pos2] = value;
}
```

java



## 12.4 Tri bulle

Il s'agit d'un tri par **permutations** ayant pour but d'amener à chaque étape à la « surface » du sous-tableau non trié (on entend par là l'élément d'indice minimum) la valeur la plus petite, appelée la **bulle**. La caractéristique de cette méthode est que les comparaisons ne se font qu'entre éléments consécutifs du tableau.

**Description de l'algorithme** Prenons pour exemple un tableau de taille 14. En partant de la fin du tableau, on le parcourt vers la gauche en comparant chaque couple de valeurs consécutives. Quand deux valeurs sont dans le désordre, on les permute. Le premier parcours s'achève lorsqu'on arrive à l'élément d'indice 0 qui contient alors la « bulle », c'est-à-dire la plus petite valeur du tableau, soit 1 dans l'exemple :

0	1	2	3	4	5	6	7	8	9	10	11	12	13
10	5	12	15	4	8	1	7	12	11	3	6	5	4
10	5	12	15	4	8	1	7	12	11	3	6	4	5
10	5	12	15	4	8	1	7	12	11	3	4	6	5
10	5	12	15	4	8	1	7	12	11	3	4	6	5
10	5	12	15	4	8	1	7	12	3	11	4	6	5
10	5	12	15	4	8	1	7	3	12	11	4	6	5
10	5	12	15	4	8	1	3	7	12	11	4	6	5
10	5	12	15	4	8	1	3	7	12	11	4	6	5
10	5	12	15	4	1	8	3	7	12	11	4	6	5
10	5	12	15	1	4	8	3	7	12	11	4	6	5
10	5	12	1	15	4	8	3	7	12	11	4	6	5
10	5	1	12	15	4	8	3	7	12	11	4	6	5
10	1	5	12	15	4	8	3	7	12	11	4	6	5
1	10	5	12	15	4	8	3	7	12	11	4	6	5

Ceci achève le premier parcours. On recommence à présent le même processus dans le sous-tableau débutant au deuxième élément, ce qui donnera le contenu suivant :

1	3	10	5	12	15	4	8	4	7	12	11	5	6
---	---	----	---	----	----	---	---	---	---	----	----	---	---

Et ainsi de suite. Au total, il y aura  $n - 1$  étapes.

**Algorithme.**

```

1 /**
   * Tri bulle d'un tableau d'entiers.
   *
4  * @param myArray
   */
   public static void bubbleSort(int[] myArray) {
7     // La bulle part de la droite,
       // en faisant remonter l'élément le plus petit,
       // et éclate à gauche.
10    for (int burstPos = 0;
        burstPos < myArray.length - 1;
        burstPos++) {
13        for (int bubblePos = myArray.length - 1;
            bubblePos > burstPos;
            bubblePos--) {
16            if (myArray[bubblePos] < myArray[bubblePos - 1]) {
                swap(myArray, bubblePos, bubblePos - 1);
            }
19        }
    }
22 }

/**
   * Échange deux valeurs dans un tableau
25 *
   * @param myArray un tableau
   * @param pos1 une position
28 * @param pos2 une (autre) position
   */
   private static void swap(int[] myArray, int pos1, int pos2) {
31     int value = myArray[pos1];
        myArray[pos1] = myArray[pos2];
        myArray[pos2] = value;
34 }

```

java

## 12.5 Cas particuliers

### Tri partiel

Parfois, un tri complet de l'ensemble n'est pas utile au problème. Il n'est nécessaire de n'avoir que les «  $k$  » plus petites (ou plus grandes) valeurs. Le tri par recherche des minima et le tri bulle sont particulièrement adaptés à cette situation ; il suffit de les arrêter plus tôt. Par contre, le tri par insertion est inefficace car il ne permet pas un arrêt anticipé.

### Sélection des meilleurs

Une autre situation courante est la suivante : un ensemble de valeurs sont traitées (lues, calculées...) et il ne faut garder que les  $k$  plus petites (ou plus grandes).

L'algorithme est plus simple à écrire si on utilise une position supplémentaire en fin de tableau. Notons aussi qu'il faut tenir compte du cas où il y a moins de valeurs

que le nombre de valeurs voulues ; c'est pourquoi on ajoute une variable indiquant le nombre exact de valeurs dans le tableau.

Exemple : on lit un ensemble de valeurs strictement positives (un 0 indique la fin de la lecture) et on ne garde que les  $k$  plus petites valeurs.

```
public static int bests(int[] smallers){
2  Scanner keyboard = new Scanner(System.in);
    int val;
    nbValues = 0;
5  val = keyboard.nextInt();
    while (val >= 0){
        nbValues = insert(val, smallers, nbValues);
8  val = keyboard.nextInt();
    }
}

11 public static int insert(int val, int[] smallers, nbValues){
    int i = nbValues - 1;
14 while ( i >= 0 && val < smallers[i]){
        smallers[i+1] = smallers[i];
        i = i - 1;
17 }
    smallers[i+1] = val;
    if (nbValues < smallers.length){
20 nbValues++;
    }
    return nbValues;
23 }
```

java

## 12.6 Références

▷ [http://interstices.info/jcms/c\\_6973/les-algorithmes-de-tri](http://interstices.info/jcms/c_6973/les-algorithmes-de-tri)

Ce site [VE04] permet de visualiser les méthodes de tris en fonctionnement. Il présente tous les tris vus en première plus quelques autres comme le « tri rapide » (« quicksort ») que vous verrez en deuxième année.



Quatrième partie

Compléments



# Chapitre 13

## Les types

« Un *type* qui se trompe en disant quelque chose de faux dit peut-être quelque chose de vrai. »

*Philippe Geluck*

En langage Java, toute donnée a un type.

Les données manipulées dans un programme peuvent être écrites dans le code du programme ou être mémorisées le temps de l'exécution de celui-ci ou encore être une entrée du programme.

- ▷ Une valeur écrite dans un programme est appelée « *littéral* ». Cette *valeur* a une valeur et un type.
- ▷ Pour qu'une valeur soit mémorisée le temps de l'exécution — ou tout du moins pendant une partie de l'exécution — elle doit être stockée dans une variable (voir section 7.4 page 84). Une variable a une valeur *variable* et un type.



### Contenu

13.1	Les types primitifs . . . . .	<b>152</b>
13.1.1	Les types primitifs numériques entiers . . . . .	152
13.1.2	Les littéraux numérique entiers (excepté <code>char</code> ) . . .	153
13.1.3	Le type primitif numérique entier particulier <code>char</code> .	155
13.1.4	Les littéraux numériques entiers <code>char</code> . . . . .	155
13.1.5	Les types primitifs numériques à virgule flottante .	156
13.1.6	Les littéraux numériques à virgule flottante . . . . .	157
13.1.7	Le type primitif booléen . . . . .	158
13.1.8	Les littéraux booléens . . . . .	158
13.2	Les types références . . . . .	<b>160</b>
13.2.1	Le type référence <code>String</code> . . . . .	160
13.2.2	Les littéraux de type <code>String</code> . . . . .	160
13.2.3	Les types références <i>tableau</i> . . . . .	161
13.2.4	Durée de vie des données sur le tas . . . . .	163

## 13.1 Les types primitifs

Il existe, en Java, 8 types primitifs : des types primitifs numériques entiers, numériques à virgule flottante, les caractères et les booléens. Voici ce que dit la grammaire :

*PrimitiveType:*

*NumericType*

**boolean**

*NumericType:*

*IntegralType*

*FloatingPointType*

*IntegralType:*

(one of)

**byte short int long char**

*FloatingPointType:*

(one of)

**float double**

Chaque type a une taille déterminée.

La figure 13.1 représente tous les types primitifs en Java. Voyons les en détails.

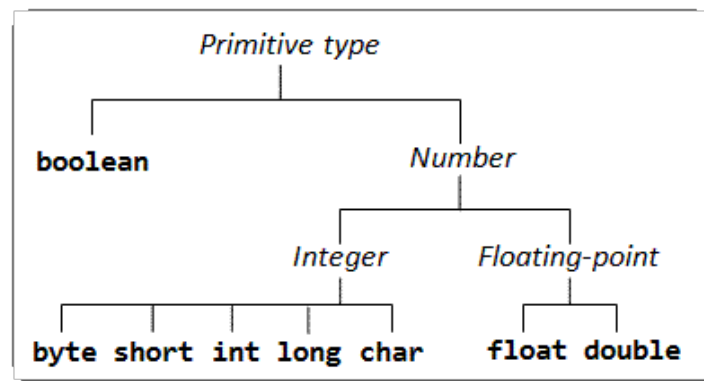


FIGURE 13.1 – Les types primitifs Java

### 13.1.1 Les types primitifs numériques entiers

Il existe 5 types primitifs entiers. Les 4 premiers représentent des nombres entiers signés codés en notation en complément à 2. Il s'agit des types : **byte**, **short**, **int** et **long**.

La quantité mathématique montrée à la figure 13.3 page ci-contre peut être représentée de différentes manières : en utilisant des chiffres romains — VII — ou arabes en base 10 — 7 — ou en base 2 — 111 — ou encore de moult autres façons.

Pour représenter un nombre négatif, l'habitude est de l'affubler du signe '-' devant mais il est également possible d'utiliser la notation en complément à 2.



Les types `byte`, `short`, `int` et `long` représentent une partie des nombres entiers signés. Ils sont stockés en notation en complément à 2. Ils se différencient par la taille qu'ils occupent en mémoire et donc par l'intervalle de nombres qu'ils représentent.

La figure 13.2 rassemble les types primitifs entiers (excepté `char`) avec leurs intervalles.

Type	Taille	Intervalle
byte	8 bits	$[-128, 127]$
short	16 bits	$[-2^7, 2^7 - 1]$ $[-32\,768, 32\,767]$
int	32 bits	$[-2^{15}, 2^{15} - 1]$ $[-2\,147\,483\,648, 2\,147\,483\,647]$
long	64 bits	$[-2^{31}, 2^{31} - 1]$ $[-9\,223\,372\,036\,854\,775\,808, 9\,223\,372\,036\,854\,775\,807]$

FIGURE 13.2 – Types primitifs entiers (exceptés `char`) et leur taille

Le type `int` est le type numérique entier privilégié. C'est celui qui est le plus utilisé et est le type par défaut dans les opérations mathématiques usuelles. L'intervalle qu'il représente est généralement suffisant <sup>61</sup>.



FIGURE 13.3 – La quantité 7 montrée avec les doigts

### 13.1.2 Les littéraux numériques entiers (excepté `char`)

Les nombres entiers sont représentés en utilisant les chiffres habituels. Il est cependant possible d'écrire un nombre dans différentes bases. Il faudra alors pouvoir les distinguer. En mathématiques il est d'habitude d'utiliser un indice pour préciser la base comme :  $7_{10} = 111_2$ . En informatique nous utiliserons un **préfixe**.

61. Ce qui n'est pas toujours vrai puisqu'en décembre 2014, la vidéo *Gangnam Style* est la première vidéo Youtube à dépasser  $\pm 2$  milliard de vues (2 147 483 647 pour être précis) et oblique Google à revoir la variable dans laquelle elle stocke ce nombre de vues pour passer à 64 bits.

### Littéral numérique décimal

Respecte les règles suivantes :

- ▷ les chiffres 0123456789 et \_ ;
- ▷ un littéral est de type `int` ou `long`, jamais de type `byte` ou `short`<sup>62</sup> ;
- ▷ pour distinguer un littéral `long` du type par défaut `int`, suffixer d'un `L` ou `L` ;

*Exemple, la quantité 100 :*

```
int myInt = 100;
int myOtherIntSomeValue = 1_00;
3 long myLong = 100L;
```

java

### Littéral numérique octal

Respecte les règles suivantes :

- ▷ les chiffres 01234567 et \_ ;
- ▷ commence par un 0 ;
- ▷ un littéral est de type `int` ou `long`, jamais de type `byte` ou `short` ;
- ▷ pour distinguer un littéral `long` du type par défaut `int`, suffixer d'un `L` ou `L` ;

*Exemple, la quantité 100 :*

```
int myOctalInt = 0144;
int anoherOctalInt = 01_44;
3 long myOctalLong = 0144L;
```

java

### Littéral numérique hexadécimal

Respecte les règles suivantes :

- ▷ les chiffres 0123456789ABCDEFabcdef et \_ ;
- ▷ commence par un `0x` ou `0X` ;
- ▷ un littéral est de type `int` ou `long`, jamais de type `byte` ou `short` ;
- ▷ pour distinguer un littéral `long` du type par défaut `int`, suffixer d'un `L` ou `L` ;

*Exemple, la quantité 100 :*

```
int myHexadecimalInt = 0x64;
long myHexadecimalLong = 0X64L;
```

java

---

62. Ceci ne nous empêchera pas d'écrire `byte b = 5` par exemple. Nous verrons les conversions en Développement II

### Littéral numérique binaire

Respecte les règles suivantes :

- ▷ les chiffres 01 et \_ ;
- ▷ commence par un 0b ou 0B ;
- ▷ un littéral est de type `int` ou `long`, jamais de type `byte` ou `short` ;
- ▷ pour distinguer un littéral long du type par défaut `int`, suffixer d'un `l` ou `L` ;

Exemple, la quantité 100 :

```
1  int myBinaryInt = 0b01100100;
   int anotherBinaryInt = 0B0110_0100;
```

java

**Remarque** Pour toutes les variables de cette section, dès lors que la quantité — 100 dans nos exemples — est stockée dans les variables : `myInt`, `myOtherInt`, `SomeValue`, `myLong` ... `anotherBinaryInt`, c'est toujours la même quantité.

Afficher toutes ses variables, affichera 100. Par exemple :

```
1  System.out.println(myBinaryInt); // 100
```

java

### 13.1.3 Le type primitif numérique entier particulier `char`

Le type `char` est un entier non signé de 16 bits représentant le code Unicode codé en UTF-16 du caractère<sup>63</sup>. Un caractère Unicode codé en UTF-16 fait une taille de 16 bits ou de 32 bits en fonction du caractère qu'il représente.

Le type `char` en Java ne permet de ne représenter que le sous ensemble BMP (*Basic Multilingual Plane*) des caractères Unicode codés en UTF-16. Ceux ayant leur code compris entre 0x0000 et 0xffff. La figure 13.7 page 159 montre quelques caractères et leurs code Unicode.

Type	Taille	Intervalle
char	16 bits	[0, 65 535] [0, 2 <sup>16</sup> - 1]

FIGURE 13.4 – Type primitif entier `char` et sa taille

### 13.1.4 Les littéraux numériques entiers `char`

Un littéral de type `char` se caractérise par les guillemet-apostrophes (*single quote* ou guillemet-simples) qui l'entourent.

Le caractère *a* par exemple, se représente `'a'`.

<sup>63</sup>. Pour en savoir plus sur l'Unicode, UTF8, UTF16 et UTF32, lire « Unicode, UTF8, UTF16, UTF32...et tutti quanti » [Bet09]

Et l'on pourra déclarer une variable de type `char` et l'initialiser avec le caractère `a` par une instruction de la forme :

```
char myChar = 'a';
```

java

La variable `myChar` de type `char` est initialisée avec le littéral `'a'` de type `char` également.



À ceci s'ajoute la possibilité **d'échapper des caractères**. Un caractère peut parfois avoir plusieurs significations. Par exemple le caractère *single quote* (`'`) signale le début d'un littéral de type `char` mais peut vouloir signaler simplement le caractère `'`. Comment distinguer les deux ?

Lorsque l'on écrit simplement un caractère, il a sa signification première, son sens premier : `a` signifie *a*, `b` signifie *b*, `Z` signifie *Z*, `'` signifie *voici le début ou la fin d'un caractère*, etc. Certains caractères ont un **deuxième sens** lorsque qu'ils sont *échappés*, c'est-à-dire précédés d'un *backslash* (`\`). La figure 13.5 montre les principaux.

Caractère	Sens premier	Sens second
<code>n</code>	<code>n</code>	passage à la ligne
<code>t</code>	<code>t</code>	tabulation
<code>'</code>	début ou fin d'un littéral <code>char</code>	<code>'</code>
<code>"</code>	début ou fin d'une chaîne	<code>"</code>
<code>\</code>	signale une séquence d'échappement (ou <i>attention, le caractère qui suit n'a pas son sens premier</i> )	<code>\</code>

FIGURE 13.5 – Quelques séquences d'échappement (la liste complète [GJS+17] section 3.10.6)

### 13.1.5 Les types primitifs numériques à virgule flottante

Les nombres pseudo-réels, ou encore les nombres à virgule flottante, sont un sous ensemble des réels parce que — comme pour les entiers — il y aura un *plus petit nombre représentable* et un *plus grand*. C'est l'**intervalle** de nombre que l'on peut représenter. À ceci, s'ajoute la **précision** que l'on pourra atteindre. En effet, les réels est un ensemble continu de nombre — entre deux nombre réels, il existe toujours, au moins, un autre réel — tandis que les pseudo-réels est un ensemble discret.

Les nombres à virgule flottante (*floating numbers*) sont codés suivant la norme IEEE 754<sup>64</sup>.

Selon cette norme, un nombre est représenté avec un signe, une mantisse et un exposant. Le tout en base 2. Un bit est utilisé pour le signe.

$$\text{nombre} = \text{signe} \text{ mantisse } 2^{\text{exposant}}$$

64. [https://fr.wikipedia.org/wiki/IEEE\\_754](https://fr.wikipedia.org/wiki/IEEE_754)

Il existe 2 types primitifs à virgule flottante : **float** et **double** dont les tailles sont données à la figure 13.6.

Type	Taille ( <i>bit</i> )	Exposant	Mantisse
float	32 bits	8	23
double	64 bits	11	52

FIGURE 13.6 – Types primitifs numériques à virgule flottante et leur taille

### 13.1.6 Les littéraux numériques à virgule flottante

Les littéraux à virgule flottante sont composés de plusieurs parties : une partie entière, un point décimal (ou hexadécimal), une partie décimale, un exposant et un suffixe.

| partie entière | . | partie décimale | exposant | suffixe |

Un littéral à virgule flottante peut être exprimé en base 10 (décimal) ou en base 16 (hexadécimal).

Un littéral à virgule flottante est de type **float** s'il est suffixé d'un **f** ou **F** sinon, il est de type **double**. Et ce qu'il soit suffixé d'un **d** ou **D** ou non.

Pour un **littéral décimal à virgule flottante**, au minimum un chiffre dans la partie entière ou décimale et soit le point décimal, soit l'exposant, soit le suffixe sont requis. Les autres parties sont optionnelles.

- ▷ la partie entière et la partie décimale sont des littéraux décimaux entiers (le caractère `_` étant autorisé) ;
- ▷ le point décimal est un point (`.`) ;
- ▷ l'exposant est la lettre **e** ou **E** suivie par un littéral décimal entier (le caractère `_` étant autorisé) ;
- ▷ le suffixe est **f**, **F**, **d** ou **D**

*Exemples :*

```
double myDouble;
2 myDouble = 1.;      // 1.0
  myDouble = .1;      // 0.1
  myDouble = 1e1;     // 10.0
5 myDouble = 1d;      // 1.0
  myDouble = 1.e0;    // 1.0
  myDouble = 1_000.45; // 1000.45
8 myDouble = 1.45e3;  // 1450.0
  myDouble = .45e3d;  // 450.0

11 float myFloat;
   myFloat = 1f;      // 1.0
   myFloat = 1.f      // 1.0
```

java

Pour un **littéral hexadécimal à virgule flottante**, au minimum un chiffre dans la partie entière ou décimale et l'exposant sont obligatoires. Le suffixe est optionnel.



- ▷ le littéral débute par 0x ou 0X;
- ▷ la partie entière et la partie décimale sont des chiffres hexadécimaux (le caractère \_ étant autorisé);
- ▷ le point décimal est un point (.);
- ▷ l'exposant est la lettre p ou P suivie par un littéral décimal entier (le caractère \_ étant autorisé) représentant une puissance de 2;
- ▷ le suffixe est f, F, d ou D

Exemples :

```
double myDouble;
2 myDouble = 0x1p0;    // 1.0 = 1 * 2^0
  myDouble = 0x1.1p0;  // 1.0625 = 1 + (1/16)
  myDouble = 0x1p1;    // 2.0 = 1 * 2^1
5 myDouble = 0xA.Bp0;  // 10.6875 = 10 + 11*1/16
  myDouble = 0x1E2p2;  // 1928 = (1*16^2 + 14*16^1 + 2*16^0) * 2^2

8 float myFloat;
  myFloat = 0X.1p4f;    // 1.0 = (1*16^-1) * 2^4
```

java

### 13.1.7 Le type primitif booléen

Le type primitif booléen permet de représenter les deux valeurs logiques *vrai* et *faux*. Ce sont les deux seules valeurs de ce type.

### 13.1.8 Les littéraux booléens

Les littéraux booléens sont simplement : *true* pour *vrai* et *false* pour *faux*.

Exemple :

```
boolean myBoolean = true;
```

java

	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	\$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
5	ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
6	ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
7	BEL 0007	ETB 0017	' 0027	7 0037	G 0047	W 0057	g 0067	w 0077
8	BS 0008	CAN 0018	( 0028	8 0038	H 0048	X 0058	h 0068	x 0078
9	HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
A	LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
B	VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[ 005B	k 006B	{ 007B
C	FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	 007C
D	CR 000D	GS 001D	- 002D	= 003D	M 004D	] 005D	m 006D	} 007D
E	SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
F	SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

FIGURE 13.7 – Table Unicode *Basic Latin* (ASCII)

## 13.2 Les types références

Pour rappel (cfr. section 7.4 page 84) une variable de type référence est une variable qui ne contient pas directement la valeur qui lui est assignée mais une référence vers cette valeur (qui se trouve à un autre endroit).

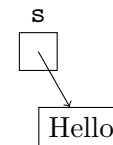
Tous les types qui ne sont pas un des 8 types primitifs présentés à la section précédente sont des types références. Ce sont les types les plus répandus bien que nous n'en verrons que quelques-uns dans ce premier cours de développement.

Dans cette section, nous parlerons essentiellement de `String` et de *tableaux* bien que l'on ait déjà rencontré d'autres types références comme `Scanner` ou `Random`. Pas d'inquiétude, nous leur réglerons leur compte plus tard, en développement II.

### 13.2.1 Le type référence `String`

`String` est un type référence.

Pour rappel, déclarer une variable de type `String` a pour effet de réserver un emplacement mémoire sur la pile (*stack*). Initialiser cette variable avec une valeur a pour effet de placer cette valeur sur le tas (*heap*). La variable *référencera* cette valeur comme nous pouvons le voir sur la figure ci-contre.



La variable `s` se trouve sur la pile.

La valeur `"Hello"` se trouve sur le tas.

Il est bien sûr possible de changer la valeur d'une variable de type référence, et dans ce cas, c'est la référence qui change. Si la valeur initiale de `s` est `Hello` et qu'elle devient `Bye bye`, la situation sera par exemple comme illustré à la figure 13.8. Nous y reviendrons.

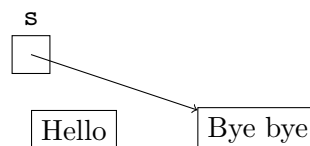


FIGURE 13.8 – Exemple : `s` reçoit `Hello` et ensuite `Bye bye`

### 13.2.2 Les littéraux de type `String`

Un littéral de type `String` se caractérise par les guillemets doubles (*double quote*) qui l'entourent (voir [GJS<sup>+</sup>17] section 3.10.5).

La chaîne `Hello` par exemple, se représente `"Hello"`. Et l'on pourra déclarer une variable de type `String` et l'initialiser avec la chaîne `Hello` par une instruction de la forme :

```
String myString = "Hello";
```

java

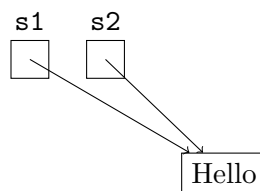


Une chaîne de caractères peut contenir n'importe quel caractère en ce compris, des caractères d'échappement. La chaîne "Hello\nWorld" a donc un sens et en cas d'affichage de cette chaîne, nous verrons un passage à la ligne entre *Hello* et *World*.

Un littéral de type `String` donné — par exemple "Hello" — référence toujours le même emplacement mémoire. Par exemple, le code suivant s'illustre par la figure 13.9.

```
String s1 = "Hello";  
2 String s2 = "Hello";
```

java

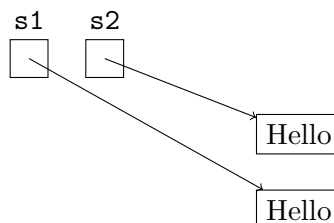
FIGURE 13.9 – Exemple : s1 et s2 reçoivent *Hello*

**Remarque** : ceci est vrai pour les littéraux de type `String` pas pour des valeurs de type `String` qui seraient, par exemple, reçues au *runtime*.

Le code suivant par exemple s'illustrerait par la figure 13.10.

```
1 Scanner keyboard = new Scanner(System.in);  
String s1 = keyboard.nextLine(); // INPUT Hello  
String s2 = keyboard.nextLine(); // INPUT Hello
```

java

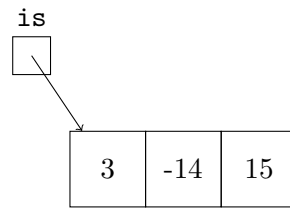
FIGURE 13.10 – Exemple : s1 et s2 reçoivent *Hello*

### 13.2.3 Les types références *tableau*

Les tableaux sont tous de types références.

Pour rappel (cfr. section 10 page 113), **déclarer** un variable de type *tableau* de a pour effet de réserver un emplacement mémoire sur la pile (*stack*). **Créer** le tableau réservera l'emplacement mémoire sur le tas (*heap*) et l'initialiser placera les valeurs dans les cases du tableau comme illustré sur la figure 13.11 page suivante.

Il est bien sûr possible de changer la valeur d'un élément du tableau, voire même de changer la valeur du tableau.

FIGURE 13.11 – Exemple : `is` est un tableau de `int` (par exemple) ayant reçu les valeurs 3, -14 et 15**Déclaration** du tableau `is`.

C'est un tableau d'entiers. Déclaré par :

```
int[] is;
```

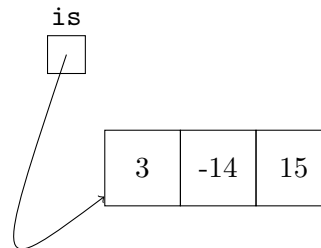
java



**Création** du tableau d'entiers et **initialisation**. La variable `is` contient une référence vers la mémoire qui contient le tableau.

```
1 int[] is = {3, -14, 15};
```

java

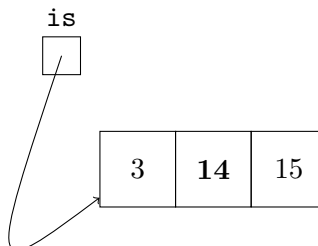


**Remplacement** d'une valeur de type primitif dans le tableau. La variable `is[1]`, de type primitif (`int`) reçoit directement une nouvelle valeur.

```
int[] is = {3, -14, 15};
is[1] = 14;
```

3

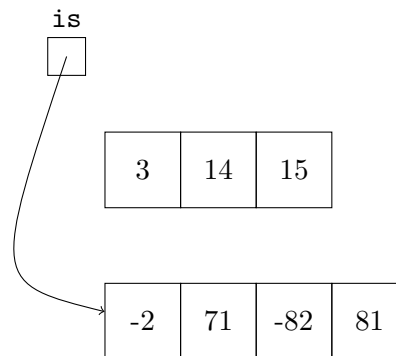
java



**Remplacement** du tableau par un autre tableau. La variable `is`, de type référence (`int[]`), reçoit une référence vers le nouveau tableau.

```
int[] is = {3, -14, 15};  
is[1] = 14;  
3 is = new int[] {-2, 71, 82,  
    81};
```

java



#### 13.2.4 Durée de vie des données sur le tas

Les données de type référence sont créées sur le tas. Chaque fois que l'on crée un tableau, une nouvelle chaîne ou une nouvelle donnée de type référence, un emplacement mémoire est réservé sur le tas pour contenir les valeurs. La variable contiendra une référence vers ces valeurs.

Nous avons vu qu'il était possible de changer la valeur d'une variable de type référence. Dans ce cas, l'ancienne valeur n'est plus référencée. Lorsque plus aucune variable ne référence une valeur en mémoire, elle n'est plus utile à personne. Comment la récupérer pour libérer de l'espace mémoire qui, bien que grand, n'est pas infini ?

Dans certains langages, c'est la tâche du développeur ou de la développeuse de *libérer* les emplacements mémoires qu'il n'utilise plus. Dans d'autres langages, c'est le langage qui s'en occupe.

En langage Java, c'est le langage qui se charge de la gestion de la mémoire. Lorsqu'il le décide, il lance un programme qui s'appelle le ramasse-miette (*garbage collector*<sup>65</sup>) qui parcourt la mémoire à la recherche des valeurs non référencées et rend les emplacements mémoires à nouveau disponibles. Il est possible de demander explicitement le passage du ramasse-miette mais ce sera la machine virtuelle qui décidera de son passage ou non.

<sup>65</sup>. Notez que la traduction n'est pas littérale.



# Chapitre 14

## Les chaînes et les `String`

Dans ce chapitre, nous allons apprendre à manipuler du texte en étudiant les différentes fonctions associées aux variables de type chaîne. Ce sera aussi l'occasion de revoir quelques techniques algorithmiques déjà acquises pour les nombres (alternatives, boucles) afin de les consolider dans un contexte plus « littéraire ».

Très tôt dans ce cours, nous avons introduit le type `chaîne` mais nous n'en avons encore fait que des usages basiques, essentiellement des affichages. Il est temps d'aller plus loin et de les *manipuler*, c'est-à-dire d'examiner et de modifier, déplacer... le contenu de chaînes.

Procédons par étapes.

### Contenu

14.1	Interface de programmation applicative . . . . .	<b>166</b>
14.2	Chaîne et caractère . . . . .	<b>166</b>
14.3	Longueur . . . . .	<b>166</b>
14.4	Le contenu d'une chaîne . . . . .	<b>167</b>
14.5	La concaténation . . . . .	<b>167</b>
14.6	Manipuler les caractères . . . . .	<b>168</b>
14.7	L'alphabet . . . . .	<b>170</b>
14.8	Chaîne et nombre . . . . .	<b>171</b>
14.9	Extraction de sous-chaînes . . . . .	<b>172</b>
14.10	Recherche de sous-chaîne . . . . .	<b>172</b>

## 14.1 Interface de programmation applicative



Une interface de programmation applicative, *application programming interface* en anglais ou encore simplement **API** est un ensemble de méthodes (de classes et de constantes) fournies — dans ce cas par le langage Java — au développeur.

L'API Java complète peut-être découverte à la lecture de la documentation fournie avec le langage à cette adresse par exemple : <https://docs.oracle.com/en/java/javase/11/docs/api/index.html> [tea18]

Un des éléments mesurant la force d'un langage est la taille de son API à laquelle s'ajoute la qualité de sa documentation. Dans cette API se retrouvent les « actions ordinaires et habituelles » que demandent les développeurs et développeuses. Intéressons-nous à la partie concernant les chaînes de caractères.

## 14.2 Chaîne et caractère

Certains langages, comme Java, distinguent clairement le type chaîne (**String**) et le type caractère (**char**). Ainsi, en Java, le littéral **'A'** représente un caractère qui est différent du littéral **"A"** qui représente une chaîne (composée d'un seul caractère). Le type **String** est un type référence tandis que le type **char** est un type primitif.

Cette distinction est essentiellement dictée par des détails d'implémentation ; un caractère pouvant se représenter de façon plus économe qu'une chaîne. En langage Java essayer de comparer — par un **"A" == 'A'** — une chaîne et un caractère génère une erreur de compilation.

D'autres langages, comme Python, n'ont pas de type caractère spécifique. D'autres encore, comme C, ont un type caractère mais pas de type chaîne à proprement parler ; ils sont vus comme des tableaux de caractères.

## 14.3 Longueur

La **longueur** d'une chaîne est le nombre de caractères qu'elle contient. Pour la connaître on utilisera la notation pointée **uneChaîne.length()**.

Par exemple :

- ▷ **"Bonjour".length()** vaut 7.
- ▷ **"Une chaîne".length()** vaut 10 (l'espace compte pour 1).
- ▷ **"A".length()** vaut 1.
- ▷ **"".length()** est égal à 0.

```
"Bonjour".length() // vaut 7
```

2

java

**Remarque** Notez la présence des parenthèses.

## 14.4 Le contenu d'une chaîne

Pour pouvoir manipuler une chaîne, il faut pouvoir accéder aux caractères qui la composent. Le langage Java, l'API, propose une méthode — `char charAt(int)` — permettant d'accéder à une lettre d'un mot et la retournant.

```
1 char c = "Hello".charAt(2);    // le caractère 'l'
```

java

**Exemple.** Écrivons un algorithme qui vérifie si un mot donné contient une lettre donnée.

```
1 public static boolean contains(String word, char letter){
    int i = 0;
    while (i < word.length() && word.charAt(i) != letter){
4       i = i + 1;
    }
    return i < word.length();
7 }
```

java

**Remarque** L'exemple est rhétorique car l'API Java propose une méthode `contains` répondant à cette question.

## 14.5 La concaténation

Il est fréquent de devoir rassembler plusieurs chaînes pour former une seule chaîne plus grande, il s'agit de l'opération de **concaténation**. En Java, c'est simplement l'opérateur `+` qui est utilisé. Dès lors que les opérandes sont des chaînes, l'opérateur concatène au lieu d'additionner.

```
1 String text;
   text = "al" + "go" + "rithmique";
```

java

**Exemple.** Écrivons un algorithme qui inverse toutes les lettres d'un mot. Ainsi, "algo" deviendra "ogla".

```
1 public static String mirror(String word){
    String mirror = "";
    for (int i = 0; i < word.length(); i = i + 1){
4       mirror = word.charAt(i) + mirror;
    }
    return mirror;
7 }
```

java

**Remarque** Cette remarque ne fait pas partie de la matière évaluée dans la partie théorique.

L'instruction de la ligne 4 concatène un caractère (`word.charAt(i)`) avec une chaîne (`mirror`) ce qui n'est, à priori, pas possible. Dans ce cas, Java convertit le caractère en une chaîne avant d'effectuer la concaténation de manière tout à fait transparente.

Dans cette instruction à la ligne 4 toujours, il y a création d'une nouvelle chaîne de caractère à chaque itération de la boucle. Si l'on veut gagner un peu de place en mémoire, il est possible de créer une chaîne de la bonne taille dès le début et d'y placer les caractères dans l'ordre qui nous convient au fur et à mesure. Ce serait mieux n'est-ce pas ?

En langage Java, ce n'est pas immédiat de remplacer un caractère par un autre dans une chaîne. Ceci parce que `Character` et `String` sont deux types différents. De plus, l'un est de type primitif comme nous l'avons déjà dit et l'autre de type référence.

Une manière élégante de résoudre le problème, est l'utilisation de `StringBuilder` qui, comme son nom l'indique, permet de manipuler plus finement une chaîne de caractères. Nous pouvons écrire :

```
1 public static String mirror(String word){
    StringBuilder sb = new StringBuilder(word);
    for (int i = 0; i < word.length(); i = i + 1){
4      sb.setCharAt(word.length() - i - 1, word.charAt(i));
    }
    return sb.toString();
7 }
```

java

- ▷ instruction 2 : création d'un espace de la même longueur que la chaîne `word` ;
- ▷ instruction 4 : placement du caractère à la position d'indice `i` dans `word` à la position d'indice « taille du mot - `i` - 1 » dans le `StringBuilder`.

## 14.6 Manipuler les caractères

Cette section ne fait pas partie de la matière évaluée dans la partie théorique.

Voici quelques méthodes de l'API Java pratiques pour la manipulation de chaînes.

Nous pourrions écrire ces algorithmes et ces méthodes de l'API mais ce serait long et fastidieux. Vous pouvez y réfléchir à titre d'exercice.

Certaines méthodes sont des méthodes de la classe `String`, d'autres de la classe `Character`. Nous vous encourageons à consulter la Javadoc [[tea18](#)].

### `isLetter`

Cette fonction indique si un caractère **est une lettre**. Par exemple elle retourne vrai pour "a", "e", "G", "K", mais faux pour "4", "\$", "@"...

boolean `Character.isLetter(char c)`



```
1 Character.isLetter('#'); // false
  char c = 'A';
  Character.isLetter(c); // true
4
```

java

### isLowerCase

Permet de savoir si le caractère **est une lettre minuscule**.

boolean Character.isLowerCase(char c)

```
Character.isLowerCase('A'); // false
2 char c = 'a';
  Character.isLowerCase(c); // true
```

java

### isUpperCase

Permet de savoir si le caractère **est une lettre majuscule**.

boolean Character.isUpperCase(char c)

```
Character.isUpperCase('a'); // false
2 char c = 'A';
  Character.isUpperCase(c); // true
```

java

### isDigit

Permet de savoir si un caractère **est un chiffre**. Elle retourne vrai pour les dix caractères '0', '1', '2', '3', '4', '5', '6', '7', '8' et '9' (et aussi pour les chiffres dans d'autres langues, cfr. javadoc<sup>66</sup>).

```
Character.isDigit('a'); // false
2 Character.isDigit('1'); // true
```

java

### toUpperCase

Convertit la chaîne en passant tous les caractères en majuscules.

String.toUpperCase()

```
String s = "HeLlO";
s.toUpperCase();
3 System.out.println(s); // affiche HELLO
```

java

### toLowerCase

Convertit la chaîne en passant tous les caractères en minuscules.

String.toLowerCase()

66. <https://docs.oracle.com/javase/9/docs/api/java/lang/Character.html#isDigit-char->

```
String s = "WOrld";  
2 s.toLowerCase();  
System.out.println(s);    // affiche world
```

java

## 14.7 L'alphabet

Cette section ne fait pas partie de la matière évaluée dans la partie théorique.

Il peut aussi être pratique de connaître la position d'une lettre dans l'alphabet pour résoudre des exercices. Cette méthode retournerait toujours un entier entre 1 et 26 (ou entre 0 et 25). Par exemple, `letterIndex('E')` donnerait 5. Cette fonction traiterait de la même manière les lettres majuscules et les lettres minuscules. Ainsi, `letterIndex('e')` retournerait également 5.

Il n'existe pas de telle méthode dans l'API Java.

Pour les caractères non accentués, il est assez simple de connaître la position d'une lettre dans l'alphabet dès lors que l'on sait qu'à chaque lettre est associée un code (Unicode) et que ces codes se suivent. La table unicode contenant les lettres de l'alphabet<sup>67</sup> nous montre que le code associé à la lettre 'A' est 0x0041 ou 65 en base 10, 'B' 0x0042 et ainsi de suite [Bet09].

Cette table nous montre également que les lettres minuscules arrivent ensuite ; 'a' a comme code 0x0061 ou 97 en base 10, 'b' 0x0062...

Retourner la position d'une lettre devient aussi simple qu'une soustraction. En supposant qu'il ne faille pas vérifier que le caractère reçu est bien une lettre, nous pourrions écrire :

```
public static int letterIndex(char c){  
2 return Character.toLowerCase(c) - 0x61 + 1;  
}
```

java

Pour traiter les caractères accentués, c'est plus complexe et ça sort un peu du cadre d'un premier cours de développement. Bien sûr, nous pouvons nous en sortir avec une série de if de la forme `if(c == 'é' || c == 'è'...)` mais ce serait fastidieux.

L'API Java peut faire ce travail pour nous si c'est demandé gentiment. Il existe une classe de l'API pour « normaliser » les caractères d'une chaîne. Cette normalisation consiste à vérifier que les caractères accentués le sont sous la forme « lettre-accent ». C'est-à-dire qu'un à est bien codé a'. Dans ce cas, `replaceAll` peut supprimer les accents et l'on pourrait écrire une méthode `letterIndex` qui fonctionne avec des lettres accentuées comme suit<sup>68</sup> :

67. <https://www.unicode.org/charts/PDF/U0000.pdf>

68. Un `import java.text.Normalizer` est nécessaire.

```

public static int letterIndex(char c){
2   c = Normalizer
    .normalize(""+c, Normalizer.Form.NFD)
    .replaceAll("[\u0300-\u036F]", "")
5   .charAt(0);
    return Character.toLowerCase(c) - 0x61 + 1;
}
8

```

java

... mais ça sort un peu du cadre de ce premier cours de développement.

Il peut être utile d'avoir un outil qui fait l'opération inverse, à savoir associer la lettre de l'alphabet correspondant à une position donnée. C'est immédiat avec une simple soustraction :

### indexToUpperChar

Retourne la forme majuscule de la  $n^{\text{e}}$  lettre de l'alphabet (où  $n$  sera obligatoirement compris entre 1 et 26). Par exemple, `indexToUpperChar(13)` retourne "M".

C'est immédiat par `return 0x41 + n - 1`.

### indexToLowerChar

Idem pour les minuscules.

C'est immédiat par `return 0x61 + n - 1`.

## 14.8 Chaîne et nombre

Cette section ne fait pas partie de la matière évaluée dans la partie théorique.

Si une chaîne contient un nombre, il doit être possible de convertir la chaîne en nombre et inversement.

### Nombre vers chaîne

La conversion d'un nombre en chaîne est immédiate lors d'une affectation dans une variable de type `String`

```

1   String s = 42;

```

java

Lors d'une concaténation, les conversions d'un nombre vers une chaîne sont immédiates à ceci près qu'il faut parfois être attentif à l'ordre des opérandes.

```

1   String s1 = 1 + "2";           // "12"
    String s2 = 1 + 2;              // "3"
    String s3 = 1 + "2" + "3";      // "123"
4   String s4 = 1 + 2 + "3";        // "33"

```

java

### Chaine vers nombre

Transforme une chaine contenant des caractères numériques en nombre. Pour chaque type, il existe une méthode qui fait cette transformation.

```
1 String s = "3.14";  
  double d = Double.parseDouble(s);  
  s = "5";  
4 int i = Integer.parseInt(s);
```

java

## 14.9 Extraction de sous-chaines

### substring

Permet d'extraire une sous-chaine d'une chaine à partir d'une position donnée jusqu'à une autre.

```
1 String s = "Karine, Jenny et Vicky paraissent belles";  
  System.out.println(s.substring(8,40));  
  // Jenny et Vicky paraissent belles  
4
```

java

Il faut bien sûr être vigilant pour ne pas sortir des bornes de la chaine sinon, une erreur est générée.

Cette fonction est très utile pour sélectionner des portions d'une chaine contenant des informations codées sous un certain format. Prenons par exemple une date stockée dans une chaine *stringDate* de format "JJ/MM/AAAA" (de longueur 10). Pour extraire :

- ▷ le jour, `substring(stringDate, 0, 2);`
- ▷ le mois, `substring(stringDate, 3, 5);`
- ▷ le jour, `substring(stringDate, 6, 10);`

## 14.10 Recherche de sous-chaine

### indexOf

Permet de savoir si une sous-chaine donnée est présente dans une chaine donnée. Elle permet d'éviter d'écrire le code correspondant à une recherche. La valeur de l'entier renvoyé est la position où commence la sous-chaine recherchée. Si la sous-chaine ne s'y trouve pas, la fonction retourne -1.

```
String s = "Des 3 nombrils, "  
2 + "c'est Karine la plus intelligente.";  
  s.indexOf("Karine"); // 22
```

java

**Remarque** Pour simplement savoir si la sous-chaine est présente sans demander sa position, la fonction `contains` retourne un booléen et prend les mêmes paramètres.



# Chapitre 15

## Les exceptions

« Attrapez-les tous, sans exception ! »

Cette section ne fait pas partie de la matière évaluée dans la partie théorique.

### Contenu

15.1	Motivation . . . . .	<b>175</b>
15.2	L'instruction try-catch . . . . .	<b>177</b>
15.3	L'instruction throw . . . . .	<b>178</b>

### 15.1 Motivation

Un programme ne tourne pas dans un monde idéal.

Il doit pouvoir *résister aux défaillances* de l'environnement, telles que :

- ▷ on tente d'ouvrir un fichier qui n'existe pas ;
- ▷ l'utilisateur entre des données incorrectes ;
- ▷ une connexion à un site web ne se fait pas ;
- ▷ etc.

Tous ces évènements *exceptionnels* viennent perturber le fonctionnement de votre programme, et doivent être gérés. Le langage Java offre un mécanisme, celui des exceptions.

À titre d'exemple, considérons le programme suivant

```

import java.util.Scanner;
2 public class Affiche {
    /**
     * Affiche un nombre entier lu au clavier.
     * @param args non utilisé
     */
    public static void main(String[] args) {
6      Scanner clavier = new Scanner(System.in);
        int nb;

11     nb = clavier.nextInt();
        System.out.println(nb);
    }
14 }

```

java

Que se passe-t-il lorsque l'utilisateur entre une lettre, ou un nombre décimal ?

```

> javac Affiche.java
> java Affiche
a
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:857)
    at java.util.Scanner.next(Scanner.java:1478)
    at java.util.Scanner.nextInt(Scanner.java:2108)
    at java.util.Scanner.nextInt(Scanner.java:2067)
    at Affiche.main(Affiche.java:7)

```

Nom de l'exception

Pile d'appels (par où il est passé)

On constate qu'une Exception a été levée (ou « lancée »), ce qui résulte en l'arrêt prématuré du programme.

### Remarque

Ici, c'est une `InputMismatchException` qui a été lancée et pas une `Exception`. En fait, nous verrons en *Développement II* qu'il existe plusieurs exceptions différentes.

Celles que nous pouvons déjà retenir sont :

#### **`InputMismatchException`**

par exemple lorsque l'utilisateur ou l'utilisatrice entre une valeur qui n'est pas du type attendu ;

#### **`ArrayIndexOutOfBoundsException`**

par exemple lorsque le programme essaie d'accéder à un élément d'un tableau hors de celui-ci ;

#### **`IllegalArgumentException`**

par exemple lorsque l'argument passé à une méthode n'est pas dans la plage des valeurs attendues ;

#### **`NullPointerException`**

par exemple lorsque l'on essaie d'accéder à un tableau qui n'a pas été créé.



## 15.2 L'instruction try-catch

Une Exception peut arrêter l'exécution du programme. Cependant, le mécanisme autorise aussi d'intercepter (ou « attraper ») les Exceptions, afin de traiter l'erreur de façon transparente pour l'utilisateur. On utilise à cet effet une instruction `try-catch`, constituée de deux blocs de code :

- ▷ `try` : contient les instructions qui *peuvent mal se passer* ;
- ▷ `catch` : contient le code qui est *en charge* de gérer le problème.

Voici la grammaire simplifiée (voir [GJS<sup>+</sup>17] page 463 pour la grammaire complète) :

```
TryStatement :
    try Block Catches

Catches :
    CatchClause {CatchClause}

CatchClause :
    catch ( Type Identifier ) Block
```

Quand une exception est levée dans un `try` :

- ▷ le code du `try` est interrompu ;
- ▷ le code du `catch` est exécuté ;
- ▷ l'exécution reprend après le bloc `try-catch`.

À titre d'exemple, on peut ainsi intercepter les exceptions levées par `nextInt()` comme ceci :

```
1 import java.util.Scanner;
   public class Affiche {
       /**
4      * Affiche l'entier lu au clavier, ou un message si ce n'est pas
        un entier.
        * @param args inutilisé.
        */
7      public static void main(String[] args) {
            Scanner clavier = new Scanner(System.in);
            int nb;
10         try {
                nb = clavier.nextInt();
                System.out.println(nb);
13         }
            catch(Exception e) {
                System.out.println("Ce_n'est_pas_un_entier!");
16         }
        }
    }
```

java

Afin d'être plus spécifique, on peut nommer dans le `catch` l'exception précise à intercepter, par exemple `InputMismatchException`.

### 15.3 L’instruction `throw`

Imaginons la situation où un programme doit demander un *entier positif* à l’utilisateur. Si le programme ne vérifie pas immédiatement que le nombre donné est entier et positif, il s’expose à des problèmes tels que plantages, résultats erronés, perte ou corruption de données, etc. avec la condition aggravante que le problème pourrait ne se signaler que plus loin dans le code, et plus tard dans le temps, rendant le débogage plus difficile. Cette constatation est à la base du mantra « Fail early, fail loudly. ».

Dès que le développeur ou la développeuse constate que son code pourrait être dans une situation instable — pourrait *planter* à court terme — à cause d’un argument par exemple, elle décide de lever une exception en utilisant l’instruction `throw` :

*ThrowStatement* :

```
throw Expression;
```

Cas pratique : vérifier les *arguments* reçus. Si un paramètre est supposé être un entier positif, une saine gestion consiste à vérifier que c’est le cas et à lever une exception sinon.

```
/**
 * Calcule la racine carrée d'un nombre.
3  * @param nb le nombre dont on veut la racine carée.
 * @return la racine carrée de <code>nb</code>.
 * @throws IllegalArgumentException si <code>nb</code> est
   négatif.
6  */
public static double racineCarrée(double nb) {
    if (nb<0) {
9      throw new IllegalArgumentException("nb_doit_être_positif!");
    }
    // Traitement normal. On est sûr que le paramètre est OK.
12 }
}
```

java

```
try {
    System.out.println( racineCarrée( val ) );
3 } catch (Exception ex) {
    System.out.println( "Calcul_impossible_" );
}
}
```

java

Comme annoncé précédemment, on peut aussi préciser qu’on n’intercepte *que* les `IllegalArgumentException`

```
1  try {  
    System.out.println( racineCarrée( val ) );  
  } catch (IllegalArgumentException ex) {  
4   System.out.println( "Calcul impossible !" );  
  }
```

java



# Chapitre 16

## La Javadoc

*« Un langage de programmation est censé être une façon conventionnelle de donner des instructions à un ordinateur, et doit pouvoir être écrit et relu par des personnes différentes.*

*Il n'est pas censé être obscur, bizarre et plein de pièges subtils.*

*Ça, ce sont les attributs de la magie. »*

*David Small*<sup>69</sup>

Cette section ne fait pas partie de la matière évaluée dans la partie théorique.

Il n'y a pas **la** documentation mais **les** documentations. Nous distinguons en effet la *documentation destinée à l'utilisateur et à l'utilisatrice* de la *documentation destinée au développeur et à la développeuse*.

### La documentation destinée à l'utilisateur et à l'utilisatrice

c'est le manuel de l'utilisateur. Ce sont des copies d'écrans et des explications du fonctionnement du programme. C'est l'ensemble des informations nécessaires à l'utilisation du programme.

### La documentation destinée au développeur et à la développeuse

c'est l'explication des méthodes que le développeur ou la développeuse peut être amenée à utiliser.

C'est ce deuxième type de documentation qui nous intéresse dans cette section.

### Contenu

16.1	La documentation . . . . .	<b>182</b>
16.2	La documentation dans le code . . . . .	<b>182</b>
16.3	Le commentaire <code>javadoc</code> . . . . .	<b>183</b>
16.3.1	Les tags . . . . .	185
16.3.2	Le style . . . . .	186

69. Cette citation date des années nonantes, dans un article critiquant le langage C (cfr. [Sma90a]) et faisant suite à une critique du langage Pascal (cfr. [Sma90b]) par un développeur Assembleur.

## 16.1 La documentation

La documentation destinée au développeur ou à la développeuse s'intéresse autant à la personne qui va utiliser le code — *je veux utiliser la méthode `Math.abs`* — qu'à la personne qui va maintenir le code — *comment avais-je écrit cette méthode de `tri` ?* —.

La personne qui va utiliser le code s'intéresse principalement à « ce que fait » la méthode. La personne qui doit maintenir le code s'intéresse bien sûr à ce que fait la méthode mais également à « comment elle le fait ». Elle a besoin de se rappeler rapidement comment elle avait fait pour *coder* la méthode pour adapter son code plus rapidement. Notons que le développeur ou la développeuse qui va la maintenir n'est peut être pas la même personne que celle qui l'a écrite la première fois. Et c'est très bien.

Penser à la documentation, c'est sans doute d'abord penser à documenter le code que l'on écrit mais c'est également penser à la documentation du code que l'on utilise. Le langage Java vient avec toute une API (cfr. section 14.1 page 166) que le développeur et la développeuse utilisent quotidiennement et que personne ne connaît par cœur et complètement. Il est donc nécessaire d'avoir la possibilité de trouver l'information rapidement et dans un format commun quelle que soit la personne qui a rédigé la documentation.

La pratique montre également que l'écriture de la documentation n'est pas la tâche préférée lorsque l'on développe. Il est donc nécessaire qu'il soit facile d'écrire et de maintenir — parce qu'il faudra également maintenir la documentation comme le code — la documentation. La documentation sera donc écrite avec le code — *literate programming* — afin de faciliter la maintenance et la synchronisation entre le code et sa documentation.

## 16.2 La documentation dans le code

Le principe de la « documentation dans le code » (*literate programming*) est très simple :

- ▷ la documentation accompagne le code ;
- ▷ un outil extrait la documentation pour en faire un document facile à lire ;
- ▷ ce document — quelle que soit la personne qui écrit la documentation — suit la même structure, le même format, le même style... et est donc facile à lire.

Coder, c'est éviter de réinventer la roue en réécrivant du code existant car assez générique. Beaucoup d'actions à exécuter sont finalement assez fréquentes : rechercher un minimum, trier, mélanger, ajouter un enregistrement dans une base de données, utiliser une liste d'éléments triés et sans doublons, envoyer un mail, créer un document au format PDF, etc. Une des tâches du développeur et de la développeuse est donc de *savoir* quelle méthode, dans quelle classe utiliser. Comme personne ne *sait* tout, une bonne développeuse ou un bon développeur est une personne qui peut

chercher et trouver rapidement. Avoir la documentation rassemblée au même endroit et écrite sous la même forme permet de gagner énormément de temps lorsque l'on cherche et lorsque que l'on lit.

La documentation commence toujours par une description de la classe complète (voir par exemple la figure 16.1) qui précise le rôle de la classe.

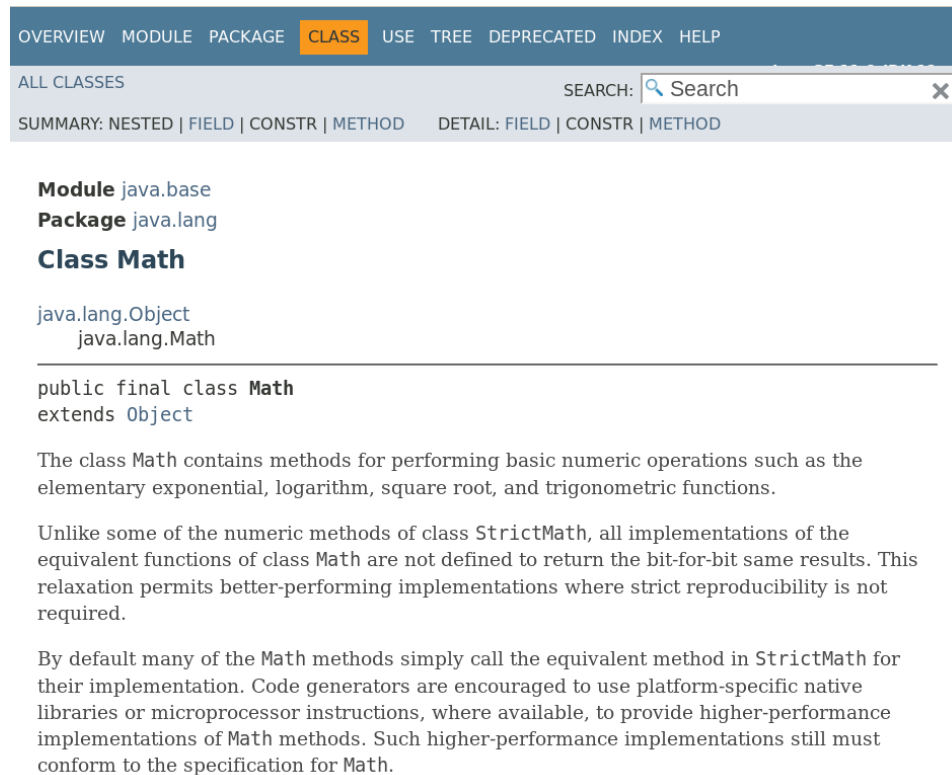


FIGURE 16.1 – Exemple : le haut de page de la documentation de la classe `Math` pour JDK11

Cette description de la classe est suivie de la liste des méthodes de la classe accompagnée de la description courte de chacune d'entre elles. Plus bas se trouve la description longue de la méthode (voir par exemple la figure 16.2 page suivante).

L'outil `javadoc` permet l'harmonisation de la documentation Java et la génération complète de la documentation.

## 16.3 Le commentaire javadoc

Comme nous l'avons vu à la section 4.6.3 page 51, il existe 3 manières d'écrire les commentaires en Java. Seuls les commentaires identifiés par `/** */` sont destinés à l'outil `javadoc`.

Le commentaire *javadoc* se place au dessus de la déclaration de la classe pour la documentation de la classe et au dessus de la déclaration de chaque méthode pour la documentation des méthodes de la classe.

Nous reviendrons sur la **déclaration d'une classe** dans le cours de Développement II<sup>70</sup>.

<sup>70</sup>. Pour la personne curieuse, la notion de déclaration d'une classe se trouve dans [GJS<sup>+</sup>17] à la section 8.1

OVERVIEW MODULE PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP Java SE 11 & JDK 11

ALL CLASSES SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

### abs

```
public static double abs(double a)
```

Returns the absolute value of a double value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned. Special cases:

- If the argument is positive zero or negative zero, the result is positive zero.
- If the argument is infinite, the result is positive infinity.
- If the argument is NaN, the result is NaN.

**API Note:**

As implied by the above, one valid implementation of this method is given by the expression below which computes a double with the same exponent and significand as the argument but with a guaranteed zero sign bit indicating a positive value:

```
Double.longBitsToDouble((Double.doubleToRawLongBits(a)<<1)>>>1)
```

**Parameters:**

a - the argument whose absolute value is to be determined

**Returns:**

the absolute value of the argument.

FIGURE 16.2 – Exemple : la description longue de la documentation de la méthode `Math.abs` pour JDK11

La **déclaration de la méthode** se compose :

- ▷ des *modifiers* comme `public` et `static` ;
- ▷ du type de retour ;
- ▷ du nom de la méthode
- ▷ de la liste de ses paramètres ;
- ▷ d'une liste éventuelles d'exceptions ;
- ▷ du corps de la méthode.

Le nom de la méthode et la liste des paramètres forment la **signature** de la méthode (cfr. section 8.4.2 de [GJS<sup>+</sup>17]). *javadoc* utilise la signature de la méthode et son type de retour pour générer la documentation. Sur base des informations contenues dans le commentaire, du type de retour de la méthode et de sa signature, *javadoc* produit des pages html ayant la même forme que celles de la documentation officielle de l'API Java.



Un commentaire comme :

```

1 /**
   * Description courte de la méthode terminée
   * par un point.
4  *
   * Un peu plus en détail ce que fait la méthode. La description
   * longue peut contenir des balises html comme <strong>écrire
7  * en gras</strong> par exemple.
   *
   * @param par1 rôle du paramètre (son type est déduit)
10 * @return valeur de retour
   */
   public static <Type> <nom>(<params>){
13   // Statement
   }

```

java

produit si l'on choisit `String` comme type de retour et un paramètre de type `int`, une documentation à l'allure de la figure 16.3,

```

static java.lang.String name(int par1)          Description courte de la méthode terminée par un point.

```

FIGURE 16.3 – Javadoc : description courte

à laquelle correspond la description longue de la figure 16.4.

```

name

public static java.lang.String name(int par1)

Description courte de la méthode terminée par un point. Un peu plus en détail ce que fait la méthode. La description
longue peut contenir des balises html comme écrire en gras par exemple.

Parameters:
par1 - rôle du paramètre (son type est visible)

Returns:
valeur de retour

```

FIGURE 16.4 – Javadoc : description longue

### 16.3.1 Les tags

Afin de mettre correctement la documentation en forme, *javadoc* utilise des **tags**. Ce sont des mots commençant par un *arobase* — @ — qui sont reconnus par la commande *javadoc*. Certains sont requis, d'autres optionnel. Certains sont destinés à la documentation de la classe<sup>71</sup> d'autres aux méthodes.

Nous retiendrons ces *tags* placés idéalement dans cet ordre :

#### @author

renseignant le ou les auteur · es de la classe. Un *tag* par auteur · e. Dans le cadre d'un développement commun, l'auteur peut être un groupe. Ce tag ne génère pas de *javadoc*, il n'est visible que dans la classe. (*classes, requis*) ;

71. Le terme « classe » est ici dans son sens large et comprend : les classes, les interfaces et les énumérations comme nous le verrons en Développement II

**@version**

comprend la date de modification et un numéro de version commençant à 1.1. (*classes, requis*);

Par exemple *1.39, 1996-01-23*

**@param**

décrivant chaque paramètre de la méthode. Il y en a un par paramètre; (*méthodes, requis*);

**@return**

décrivant ce que retourne la méthode; (*méthodes, requis* excepté si la méthode ne retourne rien);

**@throws**

listant les exceptions éventuellement lancées par la méthode; (*méthodes*);

**@see**

renseignant une autre classe ou une autre méthode relative à ce qui est documenté.

### 16.3.2 Le style

Voici quelques règles de style<sup>72</sup> pour la rédaction de la *javadoc*.

1. Seules les classes et les méthodes publiques sont commentées.
2. Dans la mesure du possible la *javadoc* s'écrit en anglais.
3. Les mots clés (*keywords*), les noms de classes, de méthodes sont écrit entre les balises `<code>` `</code>` ou dans un *inline tag* `{@code foo}`.
4. On préférera la 3<sup>e</sup> personne à la 2<sup>e</sup> personne.
5. La description d'une méthode commence par un verbe parce qu'elle implémente habituellement une opération, une action.

*Gets the label of this button.*

sera préféré à

*This method gets the label of this button.*

6. Ajouter le commentaire au nom de la méthode sans répéter le nom de la méthode. Normalement le nom de la méthode est choisi pour représenter ce qu'elle fait. Il est donc inutile de répéter ces mots dans la documentation. La documentation doit idéalement apporter une plus-value au nom de la méthode.

Éviter cette description qui ne fait que répéter les mots contenus dans le nom de la méthode.



```

1      /**
        * Sets the tool tip text.
        *
4      * @param text the text of the tool tip
        */
        public void setToolTipText(String text)
7

```

java

<sup>72</sup>. Ces règles sont principalement issues de <https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>.

Préférer cette description qui définit ce qu'est une infobulle (*tool tip*) dans un contexte plus large.

```

    /**
2      * Registers the text to display in a tool tip.
      *
      * The text displays when the cursor lingers over
5      * the component.
      *
      * @param text the string to display. If the text
8      * is null, the tool tip is turned off for
      * this component.
      */
11     public void setToolTipText(String text)

```

java

7. Le tag `@link` peut être utilisé... avec parcimonie car un lien attire l'attention. Il ne faut pas en abuser. Ce tag s'écrit dans le texte et fait référence vers l'API.
8. Lorsqu'une référence à une autre méthode est faite, on omet les parenthèses `add` ou on écrit le type des paramètres `add(int, Object)` sans nom pour les paramètres.
9. Lors de la description éventuelle d'une variable, omettre le sujet dans un souci de brièveté.

*Counter of vowels*

sera préféré à

*This is a counter of vowels.*

10. Éviter les abréviations qui pourraient ne pas être comprises par tous et toutes.

## 16.4 L'outil javadoc

Il est possible de générer la *javadoc* d'un simple clic à partir d'un IDE comme il est possible de le faire à partir d'une console par :

```
$
javadoc MyClass.java
```

terminal

Certaines options sont intéressantes :

### -d <directory>

demande de générer la documentation dans le répertoire `directory` plutôt que dans le répertoire courant. Crée le répertoire au besoin ;

### -charset utf-8

ajoute du code html permettant la prise en charge de l'utf-8 ce qui aura pour effet que les caractères accentués seront correctement affichés dans la documentation.

### -sourcepath <directory>

précise dans quels répertoires chercher les fichiers sources lorsque l'on précise

le *package* à documenter. Par défaut, cherche dans le répertoire courant. Les nom de répertoires sont séparés par deux points (:) s'il y en a plusieurs.

### Exemples :

Tous mes fichiers sources se trouvent dans le sous répertoire **src**, je génère la documentation en une seule fois dans le répertoire **doc** par :

```
$  
javadoc src/*.java -d doc -charset utf-8
```

terminal

Mes fichiers sources sont dans un répertoire **src** et dans des sous-répertoires, je peux demander à mon shell de les chercher avant d'en fournir la liste à **javadoc**. Je génère la documentation dans le répertoire **doc** par :

```
$  
javadoc $(find src -name '*.java') -d doc -charset utf-8
```

terminal

Mes classes font toutes partie du *package*<sup>73</sup> **my.package** et se trouve dans le répertoire **src/my/package/**. Je génère la documentation dans le répertoire **doc** par :

```
$  
javadoc my.package -d doc -charset utf-8 -sourcepath src
```

terminal

---

73. La notion sera vue en Développement II, à ce stade, supposent que les classes ont toutes la même instruction **package** en début de fichier

## Cinquième partie

### Annexes



# Annexe A

## Exercices

### Contenu

---

A.1	Exercices : premiers algorithmes et programmes . . . . .	<b>192</b>
A.2	Exercices : tracer un algorithme . . . . .	<b>194</b>
A.3	Exercices : division entière, quotient et reste . . . . .	<b>196</b>
A.4	Exercices : structures alternatives . . . . .	<b>199</b>
A.5	Exercices : modules et méthodes . . . . .	<b>203</b>
A.6	Exercices : structures itératives . . . . .	<b>205</b>
A.6.1	while . . . . .	205
A.6.2	for . . . . .	206
A.6.3	do while . . . . .	207
A.6.4	Exercices récapitulatifs sur les structures répétitives	207
A.7	Exercices : tableaux . . . . .	<b>211</b>
A.8	Exercices : chaîne et String . . . . .	<b>216</b>
A.9	Exercices récapitulatifs . . . . .	<b>217</b>

---

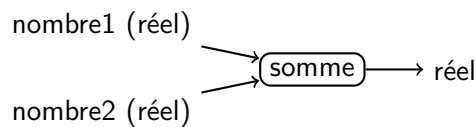
## A.1 Exercices : premiers algorithmes et programmes

Dans ces exercices, nous vous proposons d'écrire un algorithme et de le traduire en un programme Java.

### 1 Somme de 2 nombres

Calculer la somme de deux nombres donnés.

**Début de solution.**<sup>74</sup> Il y a ici clairement 2 données. Comme elles n'ont pas de rôle précis, on peut les appeler simplement `nombre1` et `nombre2` (`nb1` et `nb2` sont aussi de bons choix). L'énoncé ne dit pas si les nombres sont entiers ou pas ; restons le plus général possible en prenant des réels. Le résultat sera de même type que les données. Le nom de l'algorithme pourrait être simplement `somme`. Ce qui donne :



Et voici quelques exemples numériques :

`somme(3, 2)` donne 5    `somme(-3, 2)` donne -1

`somme(3, 2.5)` donne 5.5    `somme(-2.5, 2.5)` donne 0.

```

public class Exercices {
    public static double somme(double nombre1, double nombre2) {
3      return nombre1 + nombre2;
    }
}
  
```

java

### 2 Moyenne de 2 nombres

Calculer la moyenne de deux nombres donnés.

### 3 Surface d'un triangle

Calculer la surface d'un triangle connaissant sa base et sa hauteur.

### 4 Périmètre d'un cercle

Calculer le périmètre d'un cercle dont on donne le rayon.

### 5 Surface d'un cercle

Calculer la surface d'un cercle dont on donne le rayon.

### 6 TVA

Si on donne un prix hors TVA, il faut lui ajouter 21% pour obtenir le prix TTC. Écrire un algorithme qui permet de passer du prix HTVA au prix TTC.

<sup>74</sup>. Nous allons de temps en temps fournir des solutions. En algorithmique, il y a souvent **plusieurs** solutions possibles. Ce n'est donc pas parce que vous avez trouvé une autre façon de faire qu'elle est fausse. Mais il peut y avoir des solutions **meilleures** que d'autres ; n'hésitez jamais à montrer la vôtre à votre professeur pour avoir son avis.



**7 Les intérêts**

Calculer les intérêts reçus après 1 an pour un montant placé en banque à du 2% d'intérêt.

**8 Placement**

Étant donné le montant d'un capital placé (en €) et le taux d'intérêt annuel (en %), calculer la nouvelle valeur de ce capital après un an.

**9 Conversion HMS en secondes**

Étant donné un moment dans la journée donné par trois nombres, à savoir, heure, minute et seconde, calculer le nombre de secondes écoulées depuis minuit.

**10 Prix total**

Étant donné le prix unitaire d'un produit (hors TVA), le taux de TVA (en %) et la quantité de produit vendue à un client, calculer le prix total à payer par ce client.

**11 Durée de trajet**

Étant donné la vitesse moyenne en **m/s** d'un véhicule et la distance parcourue en **km** par ce véhicule, calculer la durée en secondes du trajet de ce véhicule.

**12 Allure et vitesse**

L'allure d'un coureur est le temps qu'il met pour parcourir 1 km (par exemple, 4'37"). On voudrait calculer sa vitesse (en km/h) à partir de son allure. Par exemple, la vitesse d'un coureur ayant une allure de 4'37" est de 13 km/h.

**13 Cote moyenne**

Étant donné les résultats (cote entière sur 20) de trois examens passés par un étudiant (exprimés par six nombres, à savoir, la cote et la pondération de chaque examen), calculer la moyenne globale exprimée en pourcentage.

## A.2 Exercices : tracer un algorithme

Nous vous proposons de tracer vos algorithmes comme présenté dans le chapitre 4 dans la section 4.3.3, page 39.

### 14 Tracer des bouts de code

Suivez l'évolution des variables pour les bouts d'algorithmes donnés.

```
1 int a, b, c;
  a = 42;
  b = 24;
4 c = a + b;
  c = c - 1;
  a = 2 * b;
7 c = c + 1;
```

java

#	a	b	c
1			
2			
3			
4			
5			
6			
7			

```
1 int a, b, c;
  double d;
  a = 2;
4 b = a*a*a;
  c = b - a*a;
  d =
    Math.sqrt(c);
7 d = d/d;
```

java

#	a	b	c
1			
2			
3			
4			
5			
6			

### 15 Calcul de vitesse

Soit le problème suivant : « Calculer la vitesse (en km/h) d'un véhicule dont on donne la durée du parcours (en secondes) et la distance parcourue (en mètres). ».

Voici *une* solution :

```
1 public static double vitesseKMH(double distanceM, double duréeS){
  double distanceKM, duréeH;
  distanceKM = 1000 * distanceM;
4  duréeH = 3600 * duréeS;
  return distanceKM / duréeH;
}
```

java

L'algorithme, s'il est correct, devrait donner une vitesse de 1 km/h pour une distance de 1000 mètres et une durée de 3600 secondes. Testez cet algorithme avec cet exemple.

```
vitesseKMH(1000, 3600);
```

java

#	
1	
2	
3	
4	
5	

Si vous trouvez qu'il n'est pas correct, voyez ce qu'il faudrait changer pour le corriger.

**16 Allure, vitesse ou cote moyenne**

Reprenez un algorithme que vous avez déjà écrit — par exemple, le problème de l'allure du coureur (voir exercice 12 page 193) ou du calcul d'une cote moyenne (voir exercice 13 page 193) — et tracez cet algorithme.

#	
1	
2	
3	
4	
5	

### A.3 Exercices : division entière, quotient et reste

#### 17 Calculs

Voici quelques petits calculs à compléter faisant intervenir la division entière et le reste. Par exemple : « 14 DIV 3 = 4 reste 2 » signifie que  $14 \text{ DIV } 3 = 4$  et  $14 \text{ MOD } 3 = 2$ .

▷ 11 DIV 3 = \_\_\_\_ reste \_\_\_\_

▷ 11 DIV \_\_\_\_ = 2 reste 3

▷ 3 DIV 11 = \_\_\_\_ reste \_\_\_\_

▷ \_\_\_\_ DIV 3 = 3 reste 1

#### 18 Les prix ronds

Voici un algorithme qui reçoit une somme d'argent exprimée en centimes et qui calcule le nombre (entier) de centimes qu'il faudrait ajouter à la somme pour tomber sur un prix rond en euros. Testez-le avec des valeurs numériques. Est-il correct ?

```
public static int versPrixRond(int prixCentimes){
2  return 100 - (prixCentimes % 100);
}
```

java

test n°	prixCentimes	réponse correcte	valeur retournée	Correct ?
1	130	70		
2	40	60		
3	99	1		
4	100	0		

**Remarque** Les exercices qui suivent n'ont pas tous été déjà analysés et ils demandent des calculs faisant intervenir des divisions entières, des restes et/ou des expressions booléennes. Comme d'habitude, écrivez la spécification si ça n'a pas encore été fait, donnez des exemples, rédigez un algorithme et vérifiez-le.

#### 19 Nombre multiple de 5

Calculer si un nombre entier donné est un multiple de 5.

**Solution.** Dans ce problème, il y a une donnée, le nombre à tester. La réponse est un booléen qui est à vrai si le nombre donné est un multiple de 5.

nombre (integer)  $\longrightarrow$  multiple5  $\longrightarrow$  boolean

**Exemples.**

▷ multiple5(4)      ▷ multiple5(15)      ▷ multiple5(0)      ▷ multiple5(-10)  
donne faux      donne vrai      donne vrai      donne vrai

La technique pour vérifier si un nombre est un multiple de 5 est de vérifier que le reste de la division par 5 donne 0. Ce qui donne :

```
return nombre % 5 == 0;
```

java

Vérifions sur nos exemples :

test n°	nombre	réponse correcte	valeur retournée	Correct ?
1	4	faux	faux	✓
2	15	vrai	vrai	✓
3	0	vrai	vrai	✓
4	-10	vrai	vrai	✓

### 20 Nombre entier positif se terminant par un 0

Calculer si un nombre donné se termine par un 0.

### 21 Les centaines

Calculer la partie *centaine* d'un nombre entier positif quelconque.

### 22 Somme des chiffres

Calculer la somme des chiffres d'un nombre entier positif inférieur à 1000.

### 23 Conversion secondes en heures

Étant donné un temps écoulé depuis minuit en secondes. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "heure".

Ex : 10000 secondes donnera 2 heures.

Aide : L'heure n'est qu'un nombre exprimé en base 60 !

### 24 Conversion secondes en minutes

Étant donné un temps écoulé depuis minuit en secondes. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "minute".

Ex : 10000 secondes donnera 46 minutes.

**25 Conversion secondes en secondes**

Étant donné un temps écoulé depuis minuit en secondes. Si on devait exprimer ce temps sous la forme habituelle (heure, minute et seconde), que vaudrait la partie "seconde".

Ex : 10000 secondes donnera 40 secondes.

**26 Un double aux dés**

Écrire un algorithme qui simule le lancer de deux dés et indique s'il y a eu un double (les deux dés montrant une face identique).

**27 Année bissextile**

Écrire un algorithme qui vérifie si une année est bissextile.

Pour rappel, les années bissextiles sont les années multiples de 4. Font exception, les multiples de 100 (sauf les multiples de 400 qui sont bien bissextiles). Ainsi 2012 et 2400 sont bissextiles mais pas 2010 ni 2100.

**28 Conversion en heures-minutes-secondes**

Écrire un algorithme qui permet à l'utilisateur de donner le nombre de secondes écoulées depuis minuit et qui affiche le moment de la journée correspondant en heures-minutes-secondes. Par exemple, si on est 3726 secondes après minuit alors il est 1h2'6".

## A.4 Exercices : structures alternatives

**Remarque** Dans certains des exercices qui suivent vous aurez besoin d'obtenir une valeur « au hasard ». Vous pouvez supposer qu'il existe une instruction `random(n)` qui donne une valeur au hasard comprise entre 0 et n (strictement).

Cette notion est abordée dans la section 4.4.5, p.46.

### 29 Compréhension

Tracez cet algorithme avec les valeurs fournies et donnez la valeur de retour.

```
1 public static int exerciceA(int a, int b){  
    int c;  
    c = 2 * a;  
4    if (c > b){  
        c = c - b;  
    }  
7    return c;  
}
```

java

▷ `exerciceA(2, 5)` = \_\_\_\_

▷ `exerciceA(4, 1)` = \_\_\_\_

### 30 Compréhension

Tracez ces algorithmes avec les valeurs fournies et donnez la valeur de retour.

```
public static int exerciceB(int a, int b){  
    int c;  
3    if (a > b){  
        c = a/b;  
    } else {  
6        c = a%b;  
    }  
    return c;  
9 }
```

java

▷ `exerciceB(2, 3)` = \_\_\_\_

▷ `exerciceB(4, 1)` = \_\_\_\_

```

public static int exerciceC(int x1, int x2){
2  boolean isBigger;
   isBigger = x1 > x2;
   if (isBigger){
5     isBigger = isBigger && x1 == 4;
   } else {
       isBigger = isBigger || x2 == 3;
8   }
   if (isBigger){
       x1 = x1 * 1000;
11  }
   return x1 + x2;
   }
14

```

java

▷ `exerciceC(2, 3)` = \_\_\_\_▷ `exerciceC(4, 1)` = \_\_\_\_

### 31 Simplification d'algorithmes

Voici deux extraits d'algorithmes que vous pouvez supposer corrects du point de vue de la syntaxe mais contenant des lignes inutiles ou des lourdeurs d'écriture. Remplacer chacune de ces portions d'algorithme par un minimum d'instructions qui auront un effet équivalent.

```

1 if (condition){
    b = true;
} else {
4  b = false;
}

```

java

```

if (n1>n2){
    b = false;
3 } else {
    if (n1<=n2){
        b = true;
6   }
}

```

java

### 32 Maximum de 2 nombres

Écrire un algorithme qui, étant donné deux nombres quelconques, recherche et retourne le plus grand des deux. Attention ! On ne veut pas savoir si c'est le premier ou le deuxième qui est le plus grand mais bien quelle est cette plus grande valeur. Le problème est donc bien défini même si les deux nombres sont identiques.

**Solution.** Une solution complète est disponible dans la fiche 4 page 236.

### 33 Calcul de salaire

Dans une entreprise, une retenue spéciale de 15% est pratiquée sur la partie du salaire mensuel qui dépasse 1200 €. Écrire un algorithme qui calcule le salaire net à partir du salaire brut. En quoi l'utilisation de constantes convient-elle pour améliorer cet algorithme ?



**34 Fonction de Syracuse**

Écrire un algorithme qui, étant donné un entier  $n$  quelconque, retourne le résultat de la fonction  $f(n) = \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n + 1 & \text{si } n \text{ est impair} \end{cases}$

**35 Tarif réduit ou pas**

Dans une salle de cinéma, le tarif plein pour une place est de 8€. Les personnes ayant droit au tarif réduit payent 7€. Écrire un algorithme qui reçoit un booléen indiquant si la personne peut bénéficier du tarif réduit et qui retourne le prix à payer.

**36 Maximum de 3 nombres**

Écrire un algorithme qui, étant donné trois nombres quelconques, recherche et retourne le plus grand des trois.

**37 Le signe**

Écrire un algorithme qui **affiche** un message indiquant si un entier est strictement négatif, nul ou strictement positif.

**38 Le type de triangle**

Écrire un algorithme qui indique si un triangle dont on donne les longueurs de ces 3 cotés est : équilatéral (tous égaux), isocèle (2 égaux) ou quelconque.

**39 Dés identiques**

Écrire un algorithme qui lance trois dés et indique si on a obtenu 3 dés de valeur identique, 2 ou aucun.

**40 Grade**

Écrire un algorithme qui retourne le grade d'un étudiant suivant la moyenne qu'il a obtenue.

Un étudiant ayant obtenu

- ▷ moins de 50% n'a pas réussi ;
- ▷ de 50% inclus à 60% exclu a réussi ;
- ▷ de 60% inclus à 70% exclu a une satisfaction ;
- ▷ de 70% inclus à 80% exclu a une distinction ;
- ▷ de 80% inclus à 90% exclu a une grande distinction ;
- ▷ de 90% inclus à 100% inclus a la plus grande distinction.

**41 Numéro du jour**

Écrire un algorithme qui retourne le numéro du jour de la semaine reçu en paramètre (1 pour "lundi", 2 pour "mardi"...).

**42 Tirer une carte**

Écrire un algorithme qui affiche l'intitulé d'une carte tirée au hasard dans un paquet de 52 cartes. Par exemple, "As de cœur", "3 de pique", "Valet de carreau" ou encore "Roi de trèfle".

**Remarque.** Il est plus facile de déterminer séparément chacune des deux caractéristiques de la carte : couleur et valeur.

**43 Nombre de jours dans un mois**

Écrire un algorithme qui retourne le nombre de jours dans un mois. Le mois est lu sous forme d'un entier (1 pour janvier...). On considère dans cet exercice que le mois de février comprend toujours 28 jours.

**44 Moyenne pondérée**

Un examen se déroule en plusieurs parties ;

- ▷ la première partie a une pondération de 5% ;
- ▷ la seconde, 25% et ;
- ▷ la dernière 70%.

Écrire un algorithme qui reçoit les 3 cotes (/20) et qui indique si l'étudiant a réussi l'examen ou non. L'algorithme affiche également la cote de l'examen.

**45 La fourchette**

Écrire un algorithme qui, étant donné trois nombres, retourne vrai si le premier des trois appartient à l'intervalle donné par le plus petit et le plus grand des deux autres (bornes exclues) et faux sinon. Qu'est-ce qui change si on inclut les bornes ?

**46 Le prix des photocopies**

Un magasin de photocopies facture 0,10 € les dix premières photocopies, 0,09 € les vingt suivantes et 0,08 € au-delà. Écrivez un algorithme qui reçoit le nombre de photocopies effectuées et qui affiche la facture correspondante.

**47 Le stationnement alternatif**

Dans une rue où se pratique le stationnement alternatif, du 1 au 15 du mois, on se gare du côté des maisons ayant un numéro impair, et le reste du mois, on se gare de l'autre côté. Écrire un algorithme qui, sur base de la date du jour et du numéro de maison devant laquelle vous vous êtes arrêté, retourne vrai si vous êtes bien stationné et faux sinon.

## A.5 Exercices : modules et méthodes

### 48 Tracer des algorithmes

Indiquer quels nombres sont successivement affichés lors de l'exécution des algorithmes ex1 et ex2.

```
1 public static void ex1(){
    int x, y;
    x = add(3, 4);
4 System.out.println(x);
    x = 3;
    y = 5;
7 y = add(x, y);
    System.out.println(y);
}
10
12 public static int add(int a, int b){
    int sum;
13 sum = a + b;
    return sum;
}
16
```

java

```
public static void ex2(){
2 int a, b;
  a = add(3, 4); // cfr. ci-dessus
  System.out.println(a);
5 a = 3;
  b = 5;
  b = sub(b, a);
8 System.out.println(b);
}
11 public static int sub(int a, int b){
    return a - b;
}
14
```

java

### 49 Appels de module

Parmi les instructions suivantes (où les variables a, b et c sont des entiers), lesquelles font correctement appel à l'algorithme d'en-tête suivant ?

```
1 public static int pgcd(int a, int b){
    // ...
}
4
```

java

- ☐ a = pgcd(24, 32)
- ☐ a = pgcd(a, 24)

- ☐  $b = 3 * \text{pgcd}(a + b, 2 * c) + 120$
- ☐  $\text{pgcd}(20, 30)$
- ☐  $a = \text{pgcd}(a, b, c)$
- ☐  $a = \text{pgcd}(a, b) + \text{pgcd}(a, c)$
- ☐  $a = \text{pgcd}(a, \text{pgcd}(a, b))$
- ☐ `System.out.println(pgcd(a, b))`
- ☐  $\text{pgcd}(a, b) = c$

### 50 Maximum de 4 nombres

Écrivez un algorithme qui calcule le maximum de 4 nombres.

### 51 Écart entre 2 durées

Étant donné deux durées données chacune par trois nombres (heure, minute, seconde), écrire un algorithme qui calcule le délai écoulé entre ces deux durées en heure(s), minute(s), seconde(s) sachant que la deuxième durée donnée est plus petite que la première.

### 52 Tirer une carte

L'exercice suivant a déjà été résolu. Refaites une solution modulaire.

Écrire un algorithme qui affiche l'intitulé d'une carte tirée au hasard dans un paquet de 52 cartes. Par exemple, "As de cœur", "3 de pique", "Valet de carreau" ou encore "Roi de trèfle".

### 53 Nombre de jours dans un mois

Écrire un algorithme qui donne le nombre de jours dans un mois. Il reçoit en paramètre le numéro du mois (1 pour janvier...) ainsi que l'année. Pour le mois de février, il faudra répondre 28 ou 29 selon que l'année fournie est bissextile ou pas. Vous devez réutiliser au maximum ce que vous avez déjà fait lors d'exercices précédents (cf. exercice 27 page 198).

### 54 Valider une date

Écrire un algorithme qui valide une date donnée par trois entiers : l'année, le mois et le jour. Vous devez réutiliser au maximum ce que vous avez déjà fait lors d'exercices précédents.

### 55 Généraliser un algorithme

Dans l'exercice 19 page 196, nous avons écrit un algorithme pour tester si un nombre est divisible par 5. Si on vous demande à présent un algorithme pour tester si un nombre est divisible par 3, vous le feriez sans peine. Idem pour tester la divisibilité par 2, 4, 6, 7, 8, 9... mais vous vous lasserez bien vite.

Écrivez un seul algorithme, plus général, qui résoud tous ces problèmes en une seule fois.

## A.6 Exercices : structures itératives

### A.6.1 while

#### 56 Compréhension d'algorithmes

Quels sont les affichages réalisés lors de l'exécution des algorithmes suivants ?

```
public static void while1(){  
2  int x;  
  x = 0;  
  while (x < 12){  
5    x = x + 2;  
  }  
  System.out.println(x);  
8 }
```

java

```
public static void while2(){  
  boolean shouldContinue = true;  
3  int x = 5;  
  while (shouldContinue) {  
    x = x + 7;  
6    shouldContinue = x % 11 != 0  
  }  
  System.out.println(x);  
9 }
```

java

```
public static void while3(){  
2  boolean isDone = false;  
  int count = 0;  
  int x = 10;  
5  while (!isDone && count < 3){  
    if (x % 2 == 0){  
      x = x + 1;  
8      isDone = x > 20;  
    } else {  
      x = x + 3;  
11     count = count + 1;  
    }  
  }  
14 System.out.println(x);  
}
```

java

#### 57 Afficher des nombres

En utilisant un **while**, écrire un algorithme qui reçoit un entier  $n$  positif et affiche

- les nombres de 1 à  $n$  ;
- les nombres de  $n$  à 1 en ordre décroissant ;

- c) les nombres impairs de 1 à  $n$  ;
- d) les nombres de  $-n$  à  $n$  ;
- e) les multiples de 5 de 1 à  $n$  ;
- f) les multiples de  $n$  de 1 à 100.

### A.6.2 for

#### 58 Compréhension d'algorithmes

Quels sont les affichages réalisés lors de l'exécution des algorithmes suivants ?

```

public static void for1(){
2   int x;
   boolean b;
   x = 3;
5   b = true;
   for (int i = 0; i < 5; i++){
       x = x + i + 1;
8       b = b && x % 2 == 0;
   }
   if (b){
11      System.out.println(x);
   } else {
       System.out.println(2 * x);
14  }
}

```

java

```

public static void for2(){
2   int end;
   for (int i = 0; i < 3; i++){
       end = 6 * (i + 1) - 11;
5       for (int j = 0; j < end; j = j + 3){
           System.out.println(10 * i + j);
       }
8   }
}

```

java

#### 59 Afficher des nombres

Reprenons un exercice déjà donné avec le **while**. En utilisant un **for**, écrire un algorithme qui reçoit un entier  $n$  positif et affiche

- a) les nombres de 1 à  $n$  ;
- b) les nombres de 1 à  $n$  en ordre décroissant ;
- c) les nombres de  $-n$  à  $n$  ;
- d) les multiples de 5 de 1 à  $n$  ;
- e) les multiples de  $n$  de 1 à 100.

### A.6.3 do while

#### 60 Compréhension d'algorithmes

Quels sont les affichages réalisés lors de l'exécution de l'algorithme suivant ?

```
public static void dowhile1(){  
2  boolean isEven, isBig;  
    int p, x;  
    x = 1;  
5  p = 1;  
    do {  
        p = 2 * p;  
8    x = x + p;  
        isEven = x % 2 == 0;  
        isBig = x > 15;  
11 } while (!isEven && !isBig);  
    System.out.println(x);
```

java

#### 61 Afficher des nombres

Reprenons un exercice déjà fait avec le **while** et le **for** en utilisant cette fois un **do while**. Écrire un algorithme qui reçoit un entier  $n$  positif et affiche

- a) les nombres de 1 à  $n$  ;
- b) les nombres de 1 à  $n$  en ordre décroissant ;
- c) les nombres de  $-n$  à  $n$  ;
- d) les multiples de 5 de 1 à  $n$  ;
- e) les multiples de  $n$  de 1 à 100.

### A.6.4 Exercices récapitulatifs sur les structures répétitives

#### 62 Afficher les nombres impairs

Écrire un algorithme qui demande une série de valeurs entières à l'utilisateur et qui affiche celles qui sont impaires. L'algorithme commence par demander à l'utilisateur le nombre de valeurs qu'il désire donner.

#### 63 Compter les nombres impairs

Écrire un algorithme qui demande une série de valeurs entières à l'utilisateur et qui lui affiche le nombre de valeurs impaires qu'il a donné. Après chaque valeur entrée, l'algorithme demande à l'utilisateur s'il y en a encore d'autres.

#### 64 Choix de la valeur sentinelle

Quelle valeur sentinelle prendrait-on pour additionner une série de cotes d'interrogations ? Une série de températures ?

#### 65 Afficher les nombres impairs

Écrire un algorithme qui demande une série de valeurs entières non nulles à l'utilisateur et qui affiche celles qui sont impaires. La fin des données sera signalée par la valeur sentinelle 0.

**66 Compter le nombre de réussites**

Écrire un algorithme qui demande une série de cotes (entières, sur 20) à l'utilisateur et qui affiche le pourcentage de réussites. La fin des données sera signalée par une valeur sentinelle que vous pouvez choisir.

**67 Suites**

Écrire les algorithmes qui affichent les  $n$  premiers termes des suites suivantes. À vous de voir quel est, parmi les 2 modèles décrits dans les exemples de la section 8.7.3 page 102, le modèle de solution le plus adapté.

- a) -1, -2, -3, -4, -5, -6...
- b) 1, 3, 6, 10, 15, 21, 28...
- c) 1, 0, 1, 0, 1, 0, 1, 0...
- d) 1, 2, 0, 1, 2, 0, 1, 2...
- e) 1, 2, 3, 1, 2, 3, 1, 2...
- f) 1, 2, 3, 2, 1, 2, 3, 2...

**Remarque** Pour les exercices qui suivent, nous vous donnons peu d'indications sur la solution à mettre en œuvre. À vous de jouer...

**68 Lire un nombre**

Écrire un algorithme qui demande à l'utilisateur un nombre entre 1 et  $n$  et le retourne. Si la valeur donnée n'est pas dans l'intervalle souhaité, l'utilisateur est invité à rentrer une nouvelle valeur jusqu'à ce qu'elle soit correcte.

**69 Lancé de dés**

Écrire un algorithme qui simule un lancé de dé. Il lance  $n$  fois un dé et compte le nombre de fois qu'une certaine valeur est obtenue.

```
/**
2 * Simule le lancé d'un dé et compte
  * le nombre de fois ou value apparait.
  *
5 * @param n le nombre de lancés
  * @param value la valeur comptée
  */
8 public static int rollDice(int n, int value){
  // ...
}
11
```

java



**70 Factorielle**

Écrire un algorithme qui retourne la factorielle de  $n$  (entier positif ou nul).

Rappel : la factorielle de  $n$ , notée  $n!$ , est le produit des  $n$  premiers entiers strictement positifs.

Par convention,  $0! = 1$ .

**71 Produit de 2 nombres**

Écrire un algorithme qui retourne le produit de deux entiers quelconques sans utiliser l'opérateur de multiplication, mais en minimisant le nombre d'opérations.

**72 Table de multiplication**

Écrire un algorithme qui affiche la table de multiplication des nombres de 1 à 10 (cf. l'exemple ci-contre).

```

1 x 1 = 1
1 x 2 = 2
...
1 x 10 = 10
2 x 1 = 2
...
10 x 9 = 90
10 x 10 = 100

```

**73 Double 6**

Écrire un algorithme qui lance de façon répétée deux dés. Il s'arrête lorsqu'il obtient un double 6 et retourne le nombre de lancers effectués.

**74 Nombre premier**

Écrire un algorithme qui vérifie si un entier positif est un **nombre premier**.

Rappel : un nombre est premier s'il n'est divisible que par 1 et par lui-même. Le premier nombre premier est 2.

**75 Nombres premiers**

Écrire un algorithme qui affiche les nombres premiers inférieurs ou égaux à un entier positif donné. Le module de cet algorithme fera appel de manière répétée mais économique à celui de l'exercice précédent.

**76 Somme de chiffres**

Écrire un algorithme qui calcule la somme des chiffres qui forment un nombre naturel  $n$ . Attention, on donne au départ **le** nombre et pas ses chiffres. Exemple : 133045 doit donner comme résultat 16, car  $1 + 3 + 3 + 0 + 4 + 5 = 16$ .

**77 Nombre parfait**

Écrire un algorithme qui vérifie si un entier positif est un **nombre parfait**, c'est-à-dire un nombre égal à la somme de ses diviseurs (sauf lui-même).

Par exemple, 6 est parfait car  $6 = 1 + 2 + 3$ . De même, 28 est parfait car  $28 = 1 + 2 + 4 + 7 + 14$ .

**78 Décomposition en facteurs premiers**

Écrire un algorithme qui affiche la décomposition d'un entier en facteurs premiers. Par exemple, 1001880 donnerait comme décomposition  $2^3 * 3^2 * 5 * 11^2 * 23$ .

**79 Nombre miroir**

Le miroir d'un nombre est le nombre obtenu en lisant les chiffres de droite à gauche. Ainsi le nombre miroir de 4209 est 9024. Écrire un algorithme qui calcule le miroir d'un nombre entier positif donné.

**80 Palindrome**

Écrire un algorithme qui vérifie si un entier donné forme un palindrome ou non. Un nombre palindrome est un nombre qui lu dans un sens (de gauche à droite) est identique au nombre lu dans l'autre sens (de droite à gauche).

Par exemple, 1047401 est un nombre palindrome.

**81 Jeu de la fourchette**

Écrire un algorithme qui simule le jeu de la fourchette. Ce jeu consiste à essayer de découvrir un nombre quelconque compris entre 1 et 100 inclus, tiré au sort par l'ordinateur. L'utilisateur a droit à huit essais maximum. À chaque essai, l'algorithme devra afficher un message indicatif « nombre donné trop petit » ou « nombre donné trop grand ». En conclusion, soit « bravo, vous avez trouvé en [nombre] essai(s) » soit « désolé, le nombre était [valeur] ».

## A.7 Exercices : tableaux

### 82 Déclarer et initialiser

Écrire un algorithme qui déclare un tableau de 100 chaînes et met votre nom dans la 3<sup>e</sup> case du tableau.

### 83 Initialiser un tableau à son indice

Écrire un algorithme qui déclare un tableau de 100 entiers et initialise chaque élément à la valeur de son indice. Ainsi, la case numéro  $i$  contiendra la valeur  $i$ .

### 84 Initialiser un tableau aux valeurs de 1 à 100

Écrire un algorithme qui déclare un tableau de 100 entiers et y met les nombres de 1 à 100.

### 85 Compréhension

Expliquez la différence entre  $\text{tab}[i] = \text{tab}[i+1]$  et  $\text{tab}[i] = \text{tab}[i]+1$ .

### 86 Trouver les entêtes

Écrire les entêtes (et uniquement les entêtes) des algorithmes qui résolvent les problèmes suivants :

- Écrire un algorithme qui inverse le signe de tous les éléments négatifs dans un tableau d'entiers.
- Écrire un algorithme qui donne le nombre d'éléments négatifs dans un tableau d'entiers.
- Écrire un algorithme qui détermine si un tableau d'entiers contient au moins un nombre négatif.
- Écrire un algorithme qui détermine si un tableau de chaînes contient une chaîne donnée en paramètre.
- Écrire un algorithme qui détermine si un tableau de chaînes contient au moins deux occurrences de la même chaîne, quelle qu'elle soit.
- Écrire un algorithme qui retourne un tableau donnant les  $n$  premiers nombres premiers, où  $n$  est un paramètre de l'algorithme.
- Écrire un algorithme qui reçoit un tableau d'entiers et retourne un tableau de booléens de la même taille où la case  $i$  indique si oui ou non le nombre reçu dans la case  $i$  est strictement positif.

### 87 Inverser le signe des éléments

Écrire un algorithme qui inverse le signe de tous les éléments négatifs dans un tableau d'entiers.

### 88 Somme

Écrire un algorithme qui reçoit en paramètre le tableau `integers` de  $n$  entiers et qui retourne la somme de ses éléments.

### 89 Nombre d'éléments d'un tableau

Écrire un algorithme qui reçoit en paramètre le tableau `doubles` de  $n$  réels et qui retourne le nombre d'éléments du tableau.

**90 Compter les éléments négatifs**

Écrire un algorithme qui donne le nombre d'éléments négatifs dans un tableau d'entiers.

**91 Y a-t-il une copie valant 20/20 ?**

1. Écrire un algorithme qui reçoit en paramètre le tableau `cotes` de  $n$  entiers représentant les cotes des étudiants et qui retourne un booléen indiquant s'il contient **au moins** une fois la valeur 20.
2. Écrire un algorithme qui reçoit en paramètre le tableau `cotes` de  $n$  entiers représentant les cotes des étudiants et qui retourne un booléen indiquant s'il contient **exactement** une fois la valeur 20.

**92 Lancers de deux dés**

Écrire un algorithme qui lance  $n$  fois deux dés et compte le nombre de fois que chaque somme apparaît. Cet algorithme retourne un tableau d'entiers ; un élément par somme.

**93 Nombre de jours dans un mois**

Écrire un algorithme qui reçoit un numéro de mois (de 1 à 12) ainsi qu'une année et donne le nombre de jours dans ce mois (en tenant compte des années bissextiles). N'hésitez pas à réutiliser des algorithmes déjà écrits.

**94 Inscription / désinscription**

1. Comment modifier l'algorithme d'inscription (cfr. 11.1 p 128) pour s'assurer qu'un étudiant ne s'inscrive pas deux fois ?
2. Comment modifier l'algorithme d'inscription pour refuser une inscription si le nombre maximal de participants est atteint en supposant que ce maximum est égal à la taille physique du tableau ?
3. Que se passerait-il avec l'algorithme de désinscription tel qu'il est si on demande à désinscrire un étudiant non inscrit ? Que suggérez-vous comme changement ?

**Remarque** Voici quelques exercices qui reprennent les différentes notions vues sur les tableaux. Pour chacun d'entre-eux :

- ▷ demandez-vous quel est le problème le plus proche déjà rencontré et voyez comment adapter la solution ;
- ▷ décomposez votre solution en plusieurs algorithmes si ça peut améliorer la lisibilité ;
- ▷ testez votre solution dans le cas général et dans les éventuels cas particuliers.

**95 Renverser un tableau**

Écrire un algorithme qui reçoit en paramètre le tableau `tabCar` de  $n$  caractères, et qui « renverse » ce tableau, c'est-à-dire qui permute le premier élément avec le dernier, le deuxième élément avec l'avant-dernier et ainsi de suite.

**96** Tableau symétrique ?

Écrire un algorithme qui reçoit en paramètre le tableau `tabChaines` de  $n$  chaînes et qui vérifie si ce tableau est symétrique, c'est-à-dire si le premier élément est identique au dernier, le deuxième à l'avant-dernier et ainsi de suite.

**97** Maximum

Écrire un algorithme qui reçoit en paramètre le tableau `tabEnt` de  $n$  entiers et qui retourne la plus **grande** valeur de ce tableau.

**98** Minimum

Écrire un algorithme qui reçoit en paramètre le tableau `tabEnt` de  $n$  entiers et qui retourne la plus **petite** valeur de ce tableau. Idem pour le minimum.

**99** Indice du maximum/minimum

Écrire un algorithme qui reçoit en paramètre le tableau `tabEnt` de  $n$  entiers et qui retourne l'indice de l'élément contenant la plus grande valeur de ce tableau. En cas d'ex-æquo, c'est l'indice le plus petit qui sera renvoyé.

Que faut-il changer pour renvoyer l'indice le plus grand ? Et pour retourner l'indice du minimum ? Réécrire l'algorithme de l'exercice précédent en utilisant celui-ci.

**100** Tableau ordonné ?

Écrire un algorithme qui reçoit en paramètre le tableau `valeurs` de  $n$  entiers et qui vérifie si ce tableau est ordonné (strictement) croissant sur les valeurs. L'algorithme retournera **vrai** si le tableau est ordonné, **faux** sinon.

**101** Remplir un tableau

On souhaite remplir un tableau de 20 éléments avec les entiers de 1 à 5, chaque nombre étant répété quatre fois. On voudrait deux variantes :

- a) D'abord une version où les nombres identiques sont groupés : d'abord tous les 1 puis tous les 2, ...

1	1	1	1	2	2	2	2	3	3	3	3	4	4
								4	4	5	5	5	5

- b) Ensuite une version où on trouve dans le tableau les valeurs 1, 2, 3, 4, 5 qui se suivent, quatre fois.

1	2	3	4	5	1	2	3	4	5	1	2	3	4
								5	1	2	3	4	5

**Remarque :** Il existe de nombreuses façons de résoudre ce problème. On peut par exemple utiliser deux boucles imbriquées. On peut aussi adapter un algorithme de génération de suites.

**102 Positions du minimum**

Écrire un algorithme qui reçoit en paramètre le tableau `cotes` de  $n$  entiers et qui affiche le ou les indice(s) des éléments contenant la valeur minimale du tableau.

- a) Écrire une première version « classique » avec deux parcours de tableau
- b) Écrire une deuxième version qui ne parcourt qu'une seule fois `cotes` en stockant dans un deuxième tableau (de quelle taille ?) les indices du plus petit élément rencontrés (ce tableau étant à chaque fois réinitialisé lorsqu'un nouveau minimum est rencontré)
- c) Écrire une troisième version qui **retourne** le tableau contenant les indices. Écrire également un algorithme qui appelle cette version puis affiche les indices reçus.

**103 Occurrence des chiffres**

Écrire un algorithme qui reçoit un nombre entier positif ou nul en paramètre et qui retourne un tableau de 10 entiers indiquant, pour chacun de ses chiffres, le nombre de fois qu'il apparaît dans ce nombre.

Ainsi, pour le nombre 10502851125, on retournera le tableau [2,3,2,0,0,3,0,0,1,0].

**104 Les doublons**

Écrire un algorithme qui vérifie si un tableau de chaînes contient au moins 2 éléments égaux.

**105 Le crible d'Ératosthène**

« Le crible d'Ératosthène est un procédé qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel donné  $N$ . L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à  $N$  tous les multiples d'un entier. En supprimant tous les multiples, à la fin il ne restera que les entiers qui ne sont multiples d'aucun entier, et qui sont donc les nombres premiers. On commence par rayer les multiples de 2, puis à chaque fois on raye les multiples du plus petit entier restant. On peut s'arrêter lorsque le carré du plus petit entier restant est supérieur au plus grand entier restant, car dans ce cas, tous les non-premiers ont déjà été rayés précédemment. » (source : Wikipédia)

Le tableau dont il est question peut être un simple tableau de booléens ; le booléen en position " $i$ " indiquant si le nombre " $i$ " est premier ou pas.

Écrire un algorithme qui reçoit un entier  $n$  et affiche tous les entiers premiers de 1 à  $n$ <sup>75</sup>.

---

75. Tester votre programme avec de « grandes » valeurs de  $n$ .

**106 Mastermind**

Dans le jeu du Mastermind, un joueur A doit trouver une combinaison de  $n$  pions de couleur, choisie et tenue secrète par un autre joueur B. Cette combinaison peut contenir éventuellement des pions de même couleur. À chaque proposition du joueur A, le joueur B indique le nombre de pions de la proposition qui sont corrects et bien placés et le nombre de pions corrects mais mal placés.

Les propositions du joueur A, ainsi que la combinaison secrète du joueur B sont contenues dans des tableaux de  $n$  composantes de type chaîne.

Écrire deux algorithmes :

- ▷ l'un qui renvoie le nombre de pions de la bonne couleur et bien placés et ;
- ▷ l'autre qui renvoie le nombre de pions de la bonne couleur et mal placés.

## A.8 Exercices : chaîne et String

### 107 Calcul de fraction

Écrire un algorithme qui reçoit une fraction sous forme de chaîne, et retourne la valeur numérique de celle-ci. Par exemple, si la fraction donnée est "5/8", l'algorithme renverra 0,625. On peut considérer que la fraction donnée est correcte, elle est composée de 2 entiers séparés par le caractère de division '/'.

### 108 Conversion de nom

Écrire un algorithme qui reçoit le nom complet d'une personne dans une chaîne sous la forme "nom, prénom" et la renvoie au format "prénom nom" (sans virgule séparatrice). Exemple : "De Groote, Jan" deviendra "Jan De Groote".

### 109 Gauche et droite

Écrire un algorithme *gauche* et un *droite* recevant un paramètre entier  $n$  qui retourne la chaîne formée respectivement des  $n$  premiers et des  $n$  derniers caractères d'une chaîne donnée.

### 110 Grammaire

Écrire un algorithme qui met un mot en « ou » au pluriel. Pour rappel, un mot en « ou » prend un « s » à l'exception des 7 mots bijou, caillou, chou, genou, hibou, joujou et pou qui prennent un « x » au pluriel. Exemple : un clou, des clous, un hibou, des hiboux. Si le mot soumis à l'algorithme n'est pas un mot en « ou », un message adéquat sera affiché.

### 111 Normaliser une chaîne

Écrire un module qui reçoit une chaîne et retourne une autre chaîne, version normalisée de la première. Par normalisée, on entend : enlever tout ce qui n'est pas une lettre et tout mettre en majuscule.  
Exemple : "Le <COBOL>, c'est la santé!" devient "LECOBOLCESTLASANTE".

### 112 Les palindromes

Cet exercice a déjà été réalisé pour des entiers, en voici 2 versions « chaîne » !

- a) Écrire un algorithme qui vérifie si un mot donné sous forme de chaîne constitue un palindrome (comme par exemple "kayak", "radar" ou "saippuakivikauppias" (marchand de pierre de savon en finnois))
- b) Écrire un algorithme qui vérifie si une phrase donnée sous forme de chaîne constitue un palindrome (comme par exemple "Esope reste ici et se repose" ou "Tu l'as trop écrasé, César, ce Port-Salut!"). Dans cette seconde version, on fait abstraction des majuscules/minuscules et on néglige les espaces et tout signe de ponctuation.



**113 Le chiffre de César**

Depuis l'antiquité, les hommes politiques, les militaires, les hommes d'affaires cherchent à garder secret les messages importants qu'ils doivent envoyer. L'empereur César utilisait une technique (on dit un *chiffrement*) qui porte à présent son nom : remplacer chaque lettre du message par la lettre qui se situe  $k$  positions plus loin dans l'alphabet (cycliquement).

Exemple : si  $k$  vaut 2, alors le texte clair "CESAR" devient "EGUCT" lorsqu'il est chiffré et le texte "ZUT" devient "BWV".

Bien sûr, il faut que l'expéditeur du message et le récepteur se soient mis d'accord sur la valeur de  $k$ .

On vous demande d'écrire un algorithme qui reçoit une chaîne ne contenant que des lettres majuscules ainsi que la valeur de  $k$  et qui retourne la version chiffrée du message.

On vous demande également d'écrire l'algorithme de déchiffrement. Cet algorithme reçoit un message chiffré et la valeur de  $k$  qui a été utilisée pour le chiffrer et retourne le message en clair.

**Remarque** Ce second algorithme est **très simple** dès lors que l'algorithme de chiffrement est écrit.

**114 Remplacement de sous-chaînes**

Écrire un algorithme qui remplace dans une chaîne donnée toutes les sous-chaînes `ch1` par la sous-chaîne `ch2`. Attention, cet exercice est plus coriace qu'il n'y paraît à première vue... Assurez-vous que votre code n'engendre pas de boucle infinie.

## A.9 Exercices récapitulatifs

La plupart des exercices qui suivent sont inspirés d'anciens examens. Ils sont assez conséquents et reprennent la plupart des notions vues dans ce cours.

**115 AEBBCLRS**

Le Scrabble est un jeu de lettres, où le but est de former des mots ayant le score le plus élevé possible. Pour calculer le score d'un mot, une valeur est associée à chaque lettre de l'alphabet. Par exemple la lettre A vaut 1, V vaut 4, J vaut 8, ... Le score d'un mot est la somme de la valeur de ses lettres<sup>76</sup>.

Par exemple le score du mot **JAVA** est  $8 + 1 + 4 + 1 = 14$ .

Les joueurs tirent chacun 7 lettres qu'ils placent sur un *chevalet*. Le but du jeu est de former des mots à partir des lettres du chevalet et de les placer sur le plateau de jeu.

---

76. Dans le vrai jeu c'est un peu plus compliqué.

### Valeur des lettres

Écrire un algorithme, `valeurLettre`, qui reçoit un caractère et retourne sa valeur. Les valeurs des lettres sont les suivantes :

- ▷ A,E,I,L,N,O,R,S,T,U : 1 point
- ▷ D,G,M : 2 points
- ▷ B,C,P : 3 points
- ▷ F,H,V : 4 points
- ▷ J,Q : 8 points
- ▷ K,W,X,Y,Z : 10 points

L'algorithme lance une erreur avec un message adéquat si le caractère passé en paramètre n'est pas une lettre majuscule.

### Score d'un mot

Écrire un algorithme, `scoreMot`, qui reçoit une chaîne de caractères, `mot`, et retourne la valeur de ce mot, c'est-à-dire la somme de la valeur de chacune de ses lettres.

**Exemple** : si l'algorithme reçoit le mot `JAVA`, il retourne 14 (=8+1+4+1).

### Mot possible

Écrire un algorithme, `motPossible`, qui reçoit un tableau de caractères, `chevalet`, et une chaîne de caractères, `mot`, et retourne vrai si les lettres du chevalet permettent d'obtenir le mot donné, et retourne faux dans le cas contraire.

**Exemple** : si l'algorithme reçoit le tableau `[A, O, V, G, G, J, L]` et le mot `"ALGO"`, l'algorithme retourne vrai. Par contre si le mot donné est `"JAVA"`, alors l'algorithme retourne faux car la lettre `A` n'apparaît pas 2 fois dans le chevalet.

**Aide** : Une solution possible est de faire une copie du chevalet et pour chaque lettre du mot de vérifier que la lettre s'y trouve, si elle s'y trouve de la remplacer par un symbole (par exemple un '-'). Pour cela on définit un algorithme `copieChevalet` permettant de copier un chevalet et un algorithme `indiceLettre` permettant de connaître l'indice d'une lettre dans un chevalet (cet algorithme retourne -1 si la lettre ne s'y trouve pas).

### Meilleur mot

Écrire un algorithme, `meilleurMot`, qui reçoit :

- ▷ un tableau de caractères, `chevalet`, qui contient les lettres disponibles pour former un mot ;
- ▷ un tableau de chaînes de caractères, `dico`, qui contient par exemple tous les mots du dictionnaire.

L'algorithme retourne le mot du dictionnaire que l'on peut obtenir avec les lettres du chevalet et qui a le score le plus élevé. Si aucun mot n'est possible avec le chevalet, l'algorithme retourne un mot vide "".

**116 Serpents et échelles**

Serpents et échelles est un jeu de société populaire, se jouant à plusieurs, consistant à déplacer les jetons sur un tableau de cases avec un dé en essayant de monter les échelles et en évitant de trébucher sur les serpents. Pour l'histoire, le jeu peut être perçu comme une représentation d'un chemin spirituel que les humains prennent pour atteindre le ciel. Avec des bons gestes, le chemin est raccourci (ce que symbolisent les échelles), tandis qu'avec de mauvais gestes, le chemin est allongé (d'où vient le symbolisme des serpents).

Extrait de Wikipedia.

**Représentation du chemin**

Nous représenterons le chemin par un tableau d'entiers. Chaque case du tableau contiendra une valeur :

- ▷ positive pour représenter une échelle et permettre d'avancer plus vite ;
- ▷ négative pour représenter le serpent et ralentir (voire reculer).

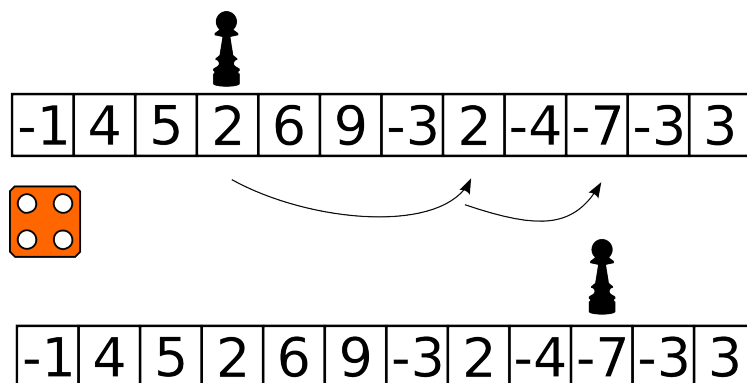
**Règles du jeu**

Pour jouer, le joueur qui a la main lance le dé et avance de la valeur indiquée par le dé. À partir de cette nouvelle position, il vérifie la valeur de la case sur laquelle il se trouve pour continuer à avancer ou au contraire reculer. Si la valeur est positive, il avance et si elle est négative, il recule de la valeur contenue dans la case.

Si la case est déjà occupée, le joueur se placera sur la case suivante.

Le joueur passe la main au joueur suivant.

**Exemple.** Le joueur se trouvant à la position 3 lance le dé. Il obtient la valeur 4. Il avance de 4 cases et se retrouve sur la case en position 7 qui contient la valeur 2. Il avance encore de 2 cases ... et termine donc son tour en position 9. Il passe alors la main au joueur suivant.



Le premier joueur à atteindre ou dépasser la dernière case a gagné. Nous supposerons que cette dernière case est en position 42. Nous supposons également que la première et la dernière case n'ont ni échelle ni serpent (valeur nulle).

### Positions des joueurs

Les positions des joueurs seront mémorisées dans un tableau d'entiers. Toutes les cases de ce tableau sont initialisées à 0.

Le plateau de jeu, un tableau d'entiers s'appellera **chemin**.

Le tableau d'entiers contenant la position courante des joueurs s'appellera **positionsJoueurs**.

S'il y a 4 joueurs, le tableau *positionsJoueurs* contient [2,5,1,7] alors :

- ▷ le joueur 0 est en position 2 sur le chemin ;
- ▷ le joueur 1 est en position 5 sur le chemin ;
- ▷ le joueur 2 est en position 1 sur le chemin ;
- ▷ le joueur 3 est en position 7 sur le chemin ;

### Le premier jour, initialiser le chemin

Écrivez un algorithme

```
1 public static int[] créerChemin(int n)
```

java

Cet algorithme *créerChemin* créera un tableau d'entiers dont les valeurs seront des valeurs aléatoires comprises strictement entre -10 et 10 (inclus).

Nous supposons que l'entier *n* est un naturel strictement supérieur à 0. Vous ne devez pas le vérifier.

### Qui est en tête ?

Écrivez un algorithme

```
1 public static int enPremièrePosition(int[] positionsJoueurs)
```

java

Cet algorithme retourne le numéro du joueur en tête de la course. Il recherche donc le maximum dans le tableau passé en argument et retourne l'indice de ce maximum.

### Jouer un tour de jeu

Cet algorithme permettra de jouer un tour de jeu pour un joueur donné. Jouer un tour de jeu consiste à le faire avancer ou reculer (c'est-à-dire changer sa position) du bon nombre de cases.

```
1 public static void jouer(int[] chemin,  
    int[] positionsJoueurs,  
    int joueurCourant,  
4    int valeurDé)
```

java

Cet algorithme fait changer de position le joueur d'indice *joueurCourant* de la valeur donnée par le dé (*valeurDé*) en respectant les règles du jeu.

Cet algorithme met donc à jour le tableau *positionsJoueurs*.

Comme le joueur ne peut se placer sur une case déjà occupée, il peut être utile d'écrire un module `estOccupé` précisant si la case est libre ou occupée.

### 117 Les algorithmes en maternelle

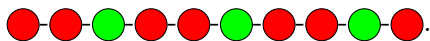
En maternelle, déjà, les enfants font des algorithmes mais il s'agit d'une chose un peu différente. On leur propose un collier<sup>77</sup> dessiné sur une feuille de papier et ils doivent le colorier en répétant un *motif* donné<sup>78</sup>, une séquence précise de couleurs.

Par exemple, on donne ce collier



qui est précolorié avec deux perles rouges et une perle verte, c'est le motif<sup>79</sup>.

Le résultat attendu est :



Comme on peut le constater sur cet exemple, un motif peut comporter plusieurs perles de la même couleur et, en fin de collier, il est possible qu'on ne puisse appliquer qu'une partie du motif.

Nous allons représenter chaque perle par un caractère indiquant sa couleur et un collier comme un tableau de caractères. Une perle non coloriée sera indiquée par un point ('.').

Par exemple, le collier ci-dessus serait représenté ainsi :

'R'	'R'	'V'	'.'	'.'	'.'	'.'	'.'	'.'	'.'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

#### Créer un collier

Écrivez un algorithme `créerCollier` qui reçoit une taille et crée un collier de cette taille où toutes les perles sont non coloriées (rappel : une perle non coloriée est représentée par un point).

On suppose que la taille reçue est bien un entier non négatif.

#### Créer un motif

Écrivez un algorithme `créerMotif` qui reçoit un collier dont aucune perle n'est coloriée et qui colorie les premières en fonction des indications de l'utilisateur.

Concrètement, l'utilisateur entre les couleurs des perles en spécifiant à chaque fois, après, s'il y a encore une perle à colorier. Dans notre exemple de la première page, l'utilisateur entrerait successivement : 'R', vrai, 'R', vrai, 'V', faux.

L'algorithme doit vérifier que l'utilisateur ne demande pas à colorier plus de perles qu'il n'y en a dans le collier.

77. Par exemple, mais ça peut aussi être une chenille, un escargot, une simple suite de cases...

78. Ce qu'ils appellent un *algorithme*.

79. Pour cet exercice, le lecteur est content s'il a la version couleurs des notes.

**Taille du motif**

Écrivez un algorithme **tailleMotif** qui reçoit un collier dont seulement les premières perles sont coloriées (c'est le motif qu'il faudra suivre) et qui donne la taille de ce motif. Dans l'exemple de la première page, il faudra retourner la valeur 3. Votre algorithme doit générer une erreur si il n'y a pas de motif à suivre.

**Suivre un motif**

Écrivez un algorithme **colorier** qui reçoit un collier dont seulement les premières perles sont coloriées (c'est le motif qu'il faut suivre) et qui colorie le reste du collier en suivant ce motif.

**Vérifier un collier**

Écrivez un algorithme **vérifier** qui reçoit un collier complètement colorié et la taille du motif de départ et qui vérifie si le collier respecte ce motif.

**Trouver le motif**

Écrivez un algorithme **trouverMotif** qui reçoit un collier complètement colorié et qui détermine la taille du motif. C'est la plus petite séquence qui se répète. Comme cas extrême, ce pourrait être le collier tout entier.

Aide : vous avez déjà tout pour que cet exercice soit facile.

# Annexe **B**

## Exercices résolus

### Contenu

B.1	Exercice résolu : le blackjack . . . . .	<b>224</b>
B.1.1	Une version simplifiée : tous les as valent 1 . . . . .	224
B.1.2	Version complète : un as vaut 1 ou 11 . . . . .	226
B.1.3	Conclusion . . . . .	227

## B.1 Exercice résolu : le blackjack

Le blackjack est un jeu de casino dans lequel le joueur ou la joueuse joue contre la banque. Le but du jeu est que la somme de ses cartes soit le plus proche de 21 sans pour autant dépasser cette valeur. Chaque carte a pour valeur son rang, sauf le valet (11), la dame (12) et roi (13) qui valent **dix** et l'as (1) qui vaut **1 ou 11**.

Écrivons un algorithme qui reçoit les cartes représentant la main du joueur ou de la joueuse dans un tableau d'entiers (de 1 à 13) et qui retourne la somme la plus avantageuse pour la personne. C'est à dire, celle qui est le plus proche de 21 sans pour autant dépasser cette valeur. Il faut donc attribuer aux as la valeur 1 ou 11 selon le jeu de la personne.

Exemples :

- ▷ pour le tableau de cartes  $[5, 12, 1, 4]$ , l'algorithme renvoie 20 ( $5 + 10$  (la dame vaut 10)  $+ 1 + 4$ ) ;
- ▷ pour le tableau de cartes  $[1, 1, 7]$  l'algorithme renvoie 19 ( $1 + 11 + 7$ ) ;
- ▷ pour le tableau de cartes  $[2, 3, 11, 1, 1]$  l'algorithme renvoie 17 ( $2 + 3 + 10 + 1 + 1$ ) ;
- ▷ pour le tableau de cartes  $[13, 3, 11, 1, 1]$  l'algorithme renvoie 25 ( $10 + 3 + 10 + 1 + 1$ ).

**Stratégie de résolution** Avant d'espérer une solution correcte à un problème, il faut s'assurer de bien le comprendre, d'identifier tous les cas particuliers et de savoir soi-même calculer la bonne réponse dans ces cas là.

Lorsque le problème semble complexe, une stratégie peut être d'attaquer d'abord une version plus simple du problème pour incorporer par la suite les éléments mis de côté. C'est ce que nous allons faire ici.

### B.1.1 Une version simplifiée : tous les as valent 1

Commençons donc par considérer une version plus simple où tous les as valent 1. Si on arrive à résoudre ce problème on réintroduira la possibilité pour un as de valoir 11.

Dans cette version simplifiée, il s'agit de calculer la somme des valeurs des cartes. Ce qu'il ne faut pas confondre avec la carte elle-même. Car si un 5 vaut 5, un 12 vaut 10.

**La somme** Comme chaque carte compte, on doit effectuer un parcours complet du tableau. On peut schématiser la solution ainsi :



Algorithme : somme simplifié d'un tableau de cartes.

Soit cartes le tableau des cartes  
Initialiser une somme à 0  
Pour toutes les cartes  
    Calculer la valeur de la carte  
    Ajouter cette valeur à la somme

*langage naturel*

Dans l'algorithme que l'on vient d'écrire, il existe un seul élément qui n'est pas trivial : le **calcul de la valeur d'une carte**. Il s'agit d'un problème en soi, qu'il est bon de regarder et de résoudre séparément sans polluer l'algorithme que l'on vient d'écrire

**La valeur d'une carte** Dans la plupart des cas, la valeur de la carte est égale à elle-même. Les seules exceptions sont les figures (valet, dame et roi) qui valent 10. On peut l'exprimer ainsi :

Algorithme : Calcul de la valeur d'une carte

Soit carte une carte  
Si la carte est > 10 alors  
    sa valeur est 10  
sinon  
    sa valeur est égale à elle-même

*langage naturel*

**Solution en Java** Si on exprime le tout en Java, ça donne :

```
1 public static int valeur(int carte) {  
    int valeur;  
    if(carte > 10) {  
4        valeur = 10;  
    } else {  
        valeur = carte;  
7    }  
    return valeur;  
}  
10  
11 public static int sommeSimplifiée(int[] cartes) {  
    int somme = 0;  
13    for(int carte : cartes) {  
        somme += valeur(carte);  
    }  
16    return somme;  
}
```

**java**

Si on veut pouvoir le tester, il faut écrire une méthode principale qui crée le tableau et appelle la méthode calculant la somme. Pour le premier exemple de l'énoncé, cela

donne

```
1 public static void main(String[] args) {
    int[] cartes = {5,12,1,4};
    int somme = sommeSimplifiée(cartes);
4   System.out.println(somme);
}
```

java

### B.1.2 Version complète : un as vaut 1 ou 11

Nous avons une solution mais rappelons-nous que nous avons simplifié le problème. Réintroduisons la règle qui dit qu'un as vaut 1 ou 11.

**1 ou 11 ? Comment décider ?** Il faut bien comprendre les points suivants :

- ▷ un as va compter pour 11 si et seulement si ça ne fait pas dépasser la valeur 21 ;
- ▷ même si plusieurs as sont présents dans la main du joueur, il ne peut y en avoir qu'un seul maximum qui compte pour 11, sinon le total dépasserait 21 ;
- ▷ on ne peut pas décider qu'un as vaut pour 11 tant qu'on n'a pas calculé la somme complète.

Par exemple avec [8,1,7], lorsqu'on rencontre le 1, on pourrait être tenté de le compter comme un 11 ( $8 + 11 = 19 < 21$ ) mais ce sera un problème par la suite pour ajouter le 7 ( $19 + 7 = 26 > 21$ ).

Des constatation ci-dessus, on peut tirer un algorithme assez simple

Algorithme : somme non simplifiée d'un tableau de cartes.

Soit cartes le tableau des cartes

Calculer la somme simplifiée, c'est à priori la bonne réponse

Mais s'il existe (au moins) un as dans les cartes

et que la somme simplifiée est  $< 12$  alors

Ajouter 10 à la somme simplifiée

langage naturel

Dans cet algorithme le seul élément non trivial qu'il reste est de savoir s'il y a (au moins) un as dans la main du joueur. Voyons comment le résoudre.

**Au moins un as** Se demander s'il y a un as dans la main du joueur (le tableau de cartes) est un problème très proche d'autres qu'on a déjà résolu (ex : "est-ce que le tableau contient un 0?"). On a vu que la solution mettait en oeuvre un **parcours partiel** du tableau. On a vu aussi qu'il y a deux solutions types : avec et sans variable booléenne. Il suffit donc de reprendre et d'adapter une des 2 approches déjà écrite par ailleurs.

**Solution en Java** La solution en Java peut utiliser les 2 méthodes déjà écrites.

```
1 public static int somme(int[] cartes) {  
    int somme = sommeSimplifiée(cartes);  
    if (existeAs(cartes) && somme<12) {  
4        somme += 10;  
    }  
    return somme;  
7 }  
  
    public static boolean existeAs(int[] cartes) {  
10    int i = 0;  
    boolean asTrouvé = false;  
    while (i<cartes.length && !asTrouvé) {  
13        asTrouvé = (cartes[i]==1);  
        i++;  
    }  
16    return asTrouvé;  
    }
```

java

Revoyez l'algorithme de parcours partiel si vous ne comprenez pas bien la méthode **existeAs**. Notez qu'on aurait pu utiliser l'autre approche pour **existeAs** ce qui aurait donné :

```
1 public static boolean existeAs(int[] cartes) {  
    int i = 0;  
    while (i<cartes.length && cartes[i]!=1) {  
4        i++;  
    }  
    return i<cartes.length;  
7 }
```

java

Vous vous rappelez pourquoi le test du return doit être celui-là ?

Il reste une dernière chose à faire : modifier la méthode principale afin qu'elle appelle la nouvelle version de la somme :

```
    public static void main(String[] args) {  
2        int[] cartes = {5,12,1,4};  
        int somme = somme(cartes);  
        System.out.println(somme);  
5    }
```

java

Ce n'est pas tout-à-fait fini. Comme tout programmeur est faillible, il reste une chose importante à faire : tester. Au minimum, en testant que le programme donne la bonne réponse dans tous les exemples donnés dans l'énoncé.

### B.1.3 Conclusion

Il y a trois enseignements à tirer de cet exercice :

1. Ne pas hésiter à attaquer d'abord une version simplifiée du problème. Cela permet d'arriver plus facilement à la solution finale.  
Ici, il n'y a même pas eu de travail inutile puisque nous avons pu réutiliser tel quel le code écrit pour la version simplifiée.
2. Décomposer la solution en plusieurs algorithmes. Si on identifie un sous-problème, il mérite une méthode à part plutôt que d'incorporer toutes les lignes de code dans une même méthode. On obtiendrait une longue méthode qui a beaucoup de défauts (plus difficile à lire/comprendre/corriger) et aucun avantage décisif (peut-être un rien plus rapide mais c'est négligeable).
3. Ne pas réinventer la roue ! Toujours se demander si le problème que l'on doit résoudre ne fait pas partie d'une catégorie de problèmes déjà résolus. Si c'est le cas, reprendre et adapter la solution déjà trouvée plutôt que de tout recommencer du début.  
Par exemple, ici, on a compris que rechercher la présence d'un as est un parcours partiel de tableau à la recherche d'un élément donné. Un problème déjà étudié et résolu.

# Annexe C

## Les fiches

Vous trouverez ici des fiches récapitulatives d'algorithmes analysés dans ce cours et présentant des problèmes habituellement rencontrés en développement.

Vous êtes invités à y faire référence.

En tant que référence — éventuellement — pour une année ultérieure, nous avons exprimé les algorithmes en pseudocode également. Lors d'une première lecture, vous pouvez laisser cette formulation de côté pour y revenir plus tard.

1	Un calcul simple . . . . .	230
2	Un calcul complexe . . . . .	232
3	Un nombre pair . . . . .	234
4	Maximum de deux nombres . . . . .	236
5	Parcours complet d'un tableau . . . . .	239
6	Maximum dans un tableau . . . . .	241
7	Parcours partiel d'un tableau . . . . .	244
8	Tableau non trié . . . . .	247
9	Tableau trié . . . . .	251
10	Recherche dichotomique . . . . .	257
11	Tri d'un tableau . . . . .	259



## n° 1

### Un calcul simple

## Le problème

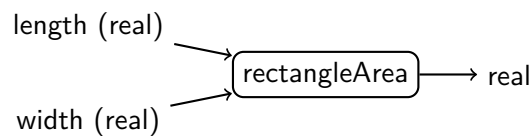
Calculer la surface d'un rectangle à partir de sa longueur et sa largeur.

## Spécification

**Données** (toutes réelles et non négatives) :

- ▷ la longueur du rectangle ;
- ▷ la largeur.

**Résultat** : un réel représentant la surface du rectangle.



## Exemples

- ▷ `rectangleArea(3, 2)` donne 6
- ▷ `rectangleArea(3.5, 1)` donne 3.5
- ▷ `rectangleArea(4, 0)` donne 0

## Solution

La surface d'un rectangle est obtenue en multipliant la largeur par la longueur.

$$\text{surface} = \text{longueur} * \text{largeur}$$

surface = longueur \* largeur

*langage naturel*

```

algorithm rectangleArea(length, width : real) → real
  return length * width
  
```

*pseudocode*

```

1 public static double rectangleArea(double length, double width){
  return length * width;
}
4
  
```

java

## Vérification / tests

test n°	longueur	largeur	réponse attendue	réponse fournie	
1	3	2	6	6	✓
2	3.5	1	3.5	3.5	✓
3	4	0	0	0	✓

## Quand l'utiliser ?

Ce type de solution peut être utilisé à chaque fois que la réponse s'obtient par un calcul simple sur les données. Si le calcul est plus complexe, il peut être utile de le décomposer pour accroître la lisibilité (cf. fiche 2 page suivante)



## n° 2

### Un calcul complexe

### Problème

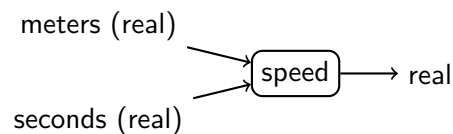
Calculer la vitesse moyenne (en km/h) d'un véhicule dont on donne la distance parcourue (en mètres) et la durée du parcours (en secondes).

### Spécification

**Données** (toutes réelles et non négatives) :

- ▷ la distance parcourue par le véhicule (en m) ;
- ▷ la durée du parcours (en s).

**Résultat** : la vitesse moyenne du véhicule (en km/h).



### Exemples

- ▷ `speed(100,10)` donne 36
- ▷ `speed(10000,3600)` donne 10

### Solution

La vitesse moyenne est liée à la distance et à la durée par la formule :

$$\text{vitesse} = \frac{\text{distance}}{\text{durée}}$$

pour autant que les unités soient cohérentes.

Ainsi pour obtenir une vitesse en km/h, il faut convertir la distance en kilomètres et la durée en heures. Ce qui donne :

convertir la distance reçue en mètres, en km : / 1000  
 convertir la durée reçue en secondes, en heure : / 3600  
 la vitesse est la distance / durée

*langage naturel*

```

algorithm speed(meters, seconds : real) → real
  real km, hours
  km = meters / 1000
  hours = seconds / 3600
  return km / hours
  
```

*pseudocode*



```

public static double speed(double meters, double seconds){
2  double km = meters / 1000;
    double hours = seconds / 3600;
    return km / hours;
5 }

```

java

## Vérification / tests

test n°	dist. (m)	durée (s)	rép. attendue	rép. fournie	
1	100	10	36	36	✓
2	10000	3600	10	10	✓

## Remarque

Nous n'avons pas tenu compte des cas où la distance serait nulle et dans ce cas, la vitesse l'est aussi et où la durée est nulle... dans ce cas la vitesse serait infinie !

## Quand l'utiliser ?

Ce type de solution peut être utilisé à chaque fois que la réponse s'obtient par un calcul complexe sur les données qu'il est bon de décomposer pour aider à sa lecture. Si le calcul est plutôt simple, on peut le garder en une seule expression (cf. fiche 1 page 230).



## n° 3

### Un nombre pair

## Problème

Un nombre reçu en paramètre est-il pair<sup>80</sup> ?

## Spécification

**Données** : le nombre entier dont on veut savoir si il est pair.

**Résultat** : un booléen à *vrai* si le *nombre* est pair et *faux* sinon.

number (integer)  $\longrightarrow$  isEven  $\longrightarrow$  boolean

## Exemples

- ▷ isEven(16) donne *vrai*
- ▷ isEven(15) donne *faux*
- ▷ isEven(0) donne *vrai*

## Solution

Un nombre est pair si il est multiple de 2. C'est-à-dire si le reste de sa division par 2 vaut 0.

n est pair si  $n \bmod 2 = 0$

*langage naturel*

```
algorithm isEven(number : integer)  $\rightarrow$  boolean
    return number MOD 2 == 0
```

*pseudocode*

```
public static boolean isEven(int n){
    return n%2 == 0;
}
```

**java**

**Remarque** C'est mieux de respecter les bonnes pratiques d'écriture et d'écrire directement un return sans if dans ce cas. Voir annexe D.3 page 265.

<sup>80</sup>. *Even* en anglais et *odd* pour impair.

## Vérification / tests

test n°	n	rép. attendue	rép. fournie	
1	16	true	true	✓
2	15	false	false	✓
3	0	true	true	✓

## Quand l'utiliser ?

À chaque fois qu'un résultat booléen dépend d'un calcul simple. Si le calcul est plus compliqué, on peut le décomposer comme indiqué dans la fiche 2 page 232.

On peut également s'inspirer de cette solution quand il faut donner sa valeur à une variable booléenne.



## n° 4

## Maximum de deux nombres

## Problème

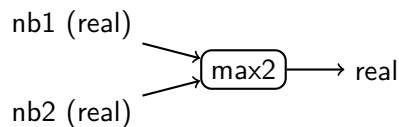
Quel est le maximum de deux nombres ?

## Analyse

Voilà un classique de l'algorithmique. Attention ! On ne veut pas savoir *lequel* est le plus grand mais juste la valeur. Il n'y a donc pas d'ambiguïté si les deux nombres sont égaux.

**Données** : deux nombres réels.

**Résultat** : un réel contenant la plus grande des deux valeurs données.



## Exemples

- ▷  $\text{max2}(-3, 4)$  donne 4
- ▷  $\text{max2}(7, 4)$  donne 7
- ▷  $\text{max2}(4, 4)$  donne 4
- ▷  $\text{max2}(-4, -8)$  donne  $-4$

## Solution

Pour trouver le maximum des deux nombres, il suffit de les comparer en utilisant un si.

```

si nb1 > nb2 alors
  nb1 est le maximum
sinon
  nb2 est le maximum
  
```

*langage naturel*

```

algorithm max2(nb1 : real, nb2 : real) → real
  real max
  if nb1 > nb2 then
    max = nb1
  else
    max = nb2
  return max
  
```

*pseudocode*

```

public static double max2(double nb1, double nb2){
2  double max;
  if (nb1 > nb2){
    max = nb1;
5  } else {
    max = nb2;
  }
8  return max;
}

```

java

Nous pourrions raisonner différemment et se dire que le maximum des deux nombres est le premier... sauf si le deuxième est plus grand. C'est une autre manière de penser.

le maximum est nb1  
 si nb2 > nb1 alors  
 le maximum est nb2

*langage naturel*

```

public static double max2(double nb1, double nb2){
2  double max = nb1
  if (nb2 > nb1){
    max = nb2;
5  }
  return max;
}
8

```

java

### Remarques

- ▷ C'est mieux de respecter les bonnes pratiques d'écriture et d'écrire directement un return sans if dans ce cas. Voir annexe D.4 page 265.
- ▷ Nous avons rapidement montré l'opérateur ternaire (cfr. Section 11.2.1 page 132) permettant d'écrire cette méthode en une seule expression.

```

1  return nb1 > nb2 ? nb1 : nb2;

```

java

### Vérification / tests

test n°	nb1	nb2	rép. attendue	rép. fournie	
1	-3	4	4	4	✓
2	7	4	7	7	✓
3	4	4	4	4	✓
4	-4	-8	-4	-4	✓

## Quand l'utiliser ?

Cet algorithme peut bien sûr être facilement adapté à la recherche du minimum.



## n° 5

### Parcours complet d'un tableau

## Problème

Parcours complet d'un tableau pour afficher tous les éléments. Par exemple un tableau de double.

## Spécification

- ▷ **Données** : le tableau à afficher
- ▷ **Résultat** : aucun.
- ▷ **Affiche** : les éléments du tableau, dans l'ordre.

## Solution

Puisqu'on parcourt tout le tableau, on peut utiliser une boucle *pour* (for).

```
algorithm display(ds : array of real)
  for i from 0 to ds.length - 1
    print ds[i]
```

*pseudocode*

```
1 // Utilisation d'un for
  public static void display (double[] ds){
    for (int i=0; i < ds.length; i++){
4      System.out.println(ds[i]);
    }
  }
7
```

java

Si l'on n'a pas besoin de modifier la valeur dans le tableau et que l'on veut simplement utiliser la valeur — dans ce cas, pour l'afficher — on peut utiliser une boucle *enhanced for* comme suit :

```
// Utilisation d'un enhanced for (foreach)
2 public static void display (double[] ds){
  for (double d: ds){
    System.out.println(d);
5  }
}
```

java

## Quand l'utiliser ?

Ce type de solution peut être utilisé à chaque fois qu'il est nécessaire d'examiner **tous** les éléments d'un tableau, quel que soit le traitement voulu : les afficher, les sommer, les comparer...

L'*enhanced for* ne permet pas de modifier les éléments du tableau.





## n° 6

### Maximum dans un tableau

#### Problème

Parcours complet d'un tableau pour rechercher la valeur maximale. Par exemple dans un tableau d'entiers.

#### Spécification

**Données** : le tableau à analyser

**Résultat** : la valeur du maximum

#### Solution

Une bonne manière de faire est d'initialiser la valeur du maximum à la première valeur du tableau et d'ensuite parcourir le reste du tableau à la recherche d'une valeur plus grande.

```
int max
int[] is
maximum = première valeur du tableau, is[0]
pour chaque valeur à partir de la deuxième
    si max > valeur alors
        max = valeur
    fin si
fin pour
```

*langage naturel*

```
algorithm max(is : array of integer) → integer
    integer max
    max = is[0]
    for i from 1 to is.length - 1
        if is[i] > max then
            max = is[i]
    return max
```

*pseudocode*

```

public static int max(int[] is){
2  int max = is[0];
  for (int i=1; i < is.length; i++){
    if (is[i] > max){
5      max = is[i];
    }
  }
8  return max;
}

```

java

Si l'on accepte de tester la première valeur, cet algorithme peut s'écrire avec un *enhanced for*.

```

public static int max(int[] is){
2  int max = is[0];
  for (value: is){
    if (value > max){
5      max = value;
    }
  }
8  return max;
}

```

java

## Variante

Si c'est la position et non la valeur qui nous intéresse, une solution serait :

```

algorithm max(is : array of integer) → integer
integer maxIndex
maxIndex = 0
for i from 1 to is.length - 1
  if is[i] > is[maxIndex] then
    maxIndex = i
return maxIndex

```

pseudocode

```
public static int max(int[] is){  
2  int maxIndex = 0;  
    for (int i=1; i < is.length; i++){  
        if (is[i] > is[maxIndex]){  
5      maxIndex = i;  
        }  
    }  
8  return max;  
}
```

[java](#)



## n° 7

## Parcours partiel d'un tableau

## Problème

Parcours partiel d'un tableau pour rechercher un zéro. Par exemple avec un tableau de double.

## Spécification

**Données** : le tableau à tester

**Résultat** : un booléen à vrai si il existe une valeur nulle dans le tableau.

ds (tableau de pseudo-réels)  $\longrightarrow$  `containsNulValue`  $\longrightarrow$  booléen

## Solution

Contrairement au parcours complet (cf. fiche 5 page 239) nous allons utiliser un *tant que* (while) car nous voulons arrêter l'algorithme dès que la valeur cherchée est trouvée.

Il existe essentiellement deux solutions, avec ou sans variable booléenne. En général, la première solution [A] sera plus claire si le test est court.

### [A] Sans variable booléenne

```

algorithm containsNulValue(ds : array of double)  $\rightarrow$  boolean
    integer i
    i = 0
    while i < ds.length AND ds[i]  $\neq$  0
        i = i + 1
    return i < n                                     // Si i < n -> arrêt prématuré
                                                    // c-à-d que l'on a trouvé un 0.

```

*pseudocode*

```

public static boolean containsNulValue(double[] ds){
2   int i = 0;
   while (i < ds.length && ds[i] != 0){
       i++;
5   }
   /* Si i < n c'est un arrêt prématuré et c'est que l'on a trouvé */
   return i < n
8 }

```

java

Il faut être attentif à **ne pas inverser** les deux parties du test sous peine de générer une erreur parce que l'on essaie d'accéder à un élément hors du tableau. Il faut

absolument vérifier que l'indice est bon avant de tester la valeur à cet indice. Se ce n'est pas clair, revoir la notion d'évaluation paresseuse (court-circuit). Voir 4.4.3 page 44.

### [B] Avec variable booléenne

```

algorithm containsNulValue(is : array of integer) → boolean
    integer i
    boolean isZero
    isZero = false
    i = 0
    while i < is.length AND NON isZero
        isZero = is[i] == 0
        i = i + 1
    return isZero

```

*pseudocode*

```

public static boolean containsNulValue(double[] ds){
    int i = 0;
    3  boolean isZero = false;
    while (i < ds.length && !isZero){
        isZero = ds[i] == 0;
    6    i++;
    }
    return isZero;
    9 }

```

java

Au sortir de la boucle, l'indice *i* ne désigne pas l'élément qui nous a permis d'arrêter mais le suivant. Si nécessaire, on peut remplacer l'intérieur de la boucle par :

```

    if (ds[i] == 0){
    2    isZero = true;
    } else {
        i++;
    5 }

```

java

Dans notre exemple, nous cherchons un élément particulier (un 0). Dans le cas où l'on vérifie si tous les éléments possèdent une certaine propriété (être positifs par exemple), nous veillerons à adapter le nom du booléen et son utilisation (par exemple un booléen appelé `areAllPositives`, initialisé à `true` avec un `...AND areAllPositives` dans le test.

Dans tous les cas, faites attention à ne pas accéder à l'élément — utiliser `ds[i]` — si vous n'êtes pas sûr de l'indice *i*. C'est particulièrement vrai après la boucle.

**Remarque** C'est mieux de respecter les bonnes pratiques d'écriture et d'écrire directement un `return` sans `if` dans ce cas. Voir annexe D.5 page 266.

## Quand l'utiliser ?

Ce type de solution peut être utilisé pour tout parcours d'un tableau où un arrêt prématuré est possible en se posant les questions suivantes :

- ▷ Est-ce que tous les éléments sont positifs ?
- ▷ Est-ce que les éléments sont triés ?
- ▷ Est-ce qu'un élément précis est présent ?
- ▷ ...



## n° 8

### Tableau non trié

## Problème

Rechercher, ajouter, supprimer des données dans un tableau non trié. Par exemple dans un tableau d'entiers

## Rechercher

### Spécification

Retourner l'indice d'une donnée trouvée dans un tableau non trié ou -1 si elle n'est pas trouvée.

Nous supposons que le tableau n'est pas rempli, l'algorithme recevra donc une valeur représentant le nombre d'éléments du tableau. Cette valeur est inférieure à la taille du tableau bien sûr.

**Données** : le tableau à analyser, le nombre d'éléments dans ce tableau (taille logique), la valeur à rechercher

**Résultat** : la position de l'élément si il est dans le tableau et -1 sinon.

## Solution

Il suffit de parcourir le tableau jusqu'à ce que l'on trouve la valeur ou bien que l'on arrive à la fin.

```
tant que l'on n'est pas à la fin et la valeur n'est pas trouvée
    incrémenter l'indice
fin tant que
si on n'est pas à la fin alors
    on a trouvé
sinon
    on n'a pas trouvé et on retourne -1
fin si
```

*langage naturel*

```
// Vérifie si un nombre est dans un tableau et donne sa position (-1 sinon)
algorithm indexValue(is↓ : array of integers,
                    nValues↓ : integer, value↓ : integer) → integer
    integer i
    i = 0
    while i < nValues AND is[i] ≠ value
    |   i = i + 1

    if i < nValues then
    |   return i
    else
    |   return -1
```

pseudocode

```
public static int indexValue(int[] is, int nValues, int value){
    int i = 0;
3   while (i < nValues && is[i] != value){
        i++;
    }
6   if (i < nValues){
        return i;
    } else {
9       return -1;
    }
}
12
```

java

## Ajouter

### Spécification

Ajouter une donnée non encore présente dans le tableau de données non triées.

**Données** : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à ajouter

**Résultat** : le tableau reçu est modifié en lui ajoutant la valeur si elle n'y était pas déjà

### Solution

Si l'algorithme reçoit le tableau, le nombre d'éléments et la valeur à ajouter, il suffit d'insérer la valeur dans le tableau... après avoir vérifier qu'elle n'est pas présente. Cette vérification peut se faire simplement en utilisant l'algorithme `indexValue`. Si celui-ci retourne -1, c'est que la valeur n'est pas présente.

```
if nValues < taille du tableau AND indexValue = -1
    ajouter la valeur à la fin du tableau
```

langage naturel



```
// Ajoute un nombre non encore présent dans le tableau.
algorithm add(is $\downarrow\uparrow$  : array of integer,
              nValues $\downarrow\uparrow$  : integer, value $\downarrow$  : integer)
  if nValues < is.length AND indexValue(is, nValues, value) = -1 then
    is[nValues] = value
    nValues = nValues + 1
```

*pseudocode*

**Remarque** En langage Java, le passage de paramètre se faisant par valeur (cfr. Chapitre 11 page 127), la taille logique du tableau doit être retournée et maintenue à jour par le code appelant. En langage Java, cet algorithme aurait l'allure suivante :

```
public static int add(int[] is, int nValues, int value){
  if (nValues < is.length()
3    && indexValue(is, nValues, value) == -1){
    is[nValues] = value;
    nValues = nValues + 1;
6  }
  return nValues;
}
9
```

**java**

## Supprimer

### Spécification

Supprimer une donnée d'un tableau de données non triées.

**Données** : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à supprimer

**Résultat** : le tableau reçu est modifié en lui supprimant la valeur

### Solution

Comme le tableau n'est pas trié, pour supprimer une valeur, il suffit de trouver l'index de cette valeur et de placer la dernière valeur à sa place.

```
index = l'index de la valeur à supprimer
if index != 1
  is[index] = is[nValues-1]
  nValues-
end if
```

*langage naturel*

// Supprime un nombre donné dans le tableau.

**algorithm** *delete*(  $is_{\downarrow\uparrow}$  : array of n integer,  $nValues_{\downarrow\uparrow}$  : integer,  $value_{\downarrow}$  : integer )

```
integer index
index = indexValue(is, nValues, value)
if index  $\neq$  -1 then
    is[index] = is[nValues-1]
    nValues = nValues - 1
```

*pseudocode*

```
public static int delete(int[] is, int nValues, int value){
    int index = indexValue(is, nValues, value);
3   if (index != -1){
        is[index] = is[nValues];
        nValues = nValues - 1;
6   }
    return nValues;
9   }
```

java



## n° 9 Tableau trié

### Problème

Rechercher, ajouter, supprimer des données triées dans un tableau trié. Par exemple dans un tableau d'entiers.

### Rechercher

#### Spécification

Rechercher la position où a été trouvé l'élément ou la position où il aurait dû être.

**Données** : le tableau à analyser, le nombre d'éléments dans ce tableau, la valeur à rechercher

**Résultat** : la position où a été trouvée la valeur ou la position où elle aurait dû être

### Solution

L'algorithme retourne la position où a été trouvée la valeur **ou** celle où elle devrait être. En parcourant le tableau, il suffit de s'arrêter dès que l'élément du tableau est plus grand ou égal à la valeur recherchée.

```
// Recherche un élément.
// - index : indique la position où a été trouvée la valeur
// ou la position où elle aurait dû être
algorithm findIndex( is↓ : array of integer, nValues↓ : integer,
                    value↓ : integer, ) → index integer

    integer index = 0
    while index < nValues AND is[index] < value
    |   index = index + 1

    return index
```

*pseudocode*

```
public static int findIndex(int[] is, int nValues, int value){
    int index = 0;
3   while (index < nValues && is[index] < value){
        index++;
    }
6   return index;
}
```

java

## Ajouter

### Spécification

Ajouter une donnée non encore présente dans le tableau de données triées.

**Données** : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à ajouter

**Résultat** : le tableau reçu est modifié en lui ajoutant la valeur si elle n'y était pas déjà

### Solution

Pour faire un ajout sans doublon dans un tableau triée, il faut :

1. vérifier que la valeur n'est pas déjà présente ;
2. rechercher l'endroit où elle devrait être placée ;
3. déplace les valeurs plus grande d'une position vers la droite ;
4. insère la valeur à sa place.

En décomposant le problème, voici une solution :

```

// Ajouter un nombre donné.
algorithm add( is $\downarrow\uparrow$  : array of integer, nValues $\downarrow$  : integer,
                                     value $\downarrow$  : integer )  $\rightarrow$  integer

    integer index
    boolean isFound
    if verify(is, nValues, value) == -1 then
        index = findIndex(is, nValues, value)
        shiftRight(is, index, nValues)
        is[index] = value
        nValues = nValues + 1

    return nValues

// Vérifie si un nombre est dans un tableau d'entiers trié
// et donne sa position (-1 si non présent)
algorithm verify( is $\downarrow$  : array of integers, nValues $\downarrow$  : integer,
                                     value $\downarrow$  : integer )  $\rightarrow$  integer

    integer index
    boolean isFound
    index = findIndex( is, nValues, value)
    if is[index] == value then
        return index
    else
        return -1

// Décale d'une position à droite les éléments
// entre la position début et fin
algorithm shiftRight( is $\downarrow\uparrow$  : array of integer, begin $\downarrow$  : integer, end $\downarrow$  : integer )
    for i from end to begin by -1
        is[i+1] = is[i]

```

*pseudocode*

```

1 /**
   * Ajoute un nombre s'il n'est pas présent.
   *
4  * @param is le tableau d'éléments
   * @param nValues le nombre d'éléments du tableau (taille logique)
   * @param value la valeur à insérer
7  */
   public static int add(int[] is, int nValues, int value){
       int index;
10  if (verify(is, nValues, value) == -1){
           index = findIndex(is, nValues, value);
           shiftRight(is, index, nValues);
13  is[index] = value;
           nValues++;
       }
16  return nValues;
   }

19 /**
   * Vérifie si la valeur est dans le tableau.
   *
22  * @param is le tableau d'éléments
   * @param nValues le nombre d'éléments du tableau (taille logique)
   * @param value la valeur à insérer
25  */
   public static int verify(int[] is, int nValues, int value){
       int index = findIndex(is, nValues, value);
28  if (is[index] == value) {
           return index;
       } else {
31  return -1;
       }
   }

34  /**
   * Décale d'une position vers la droite les éléments
37  * entre position début et fin.
   *
   * @param is le tableau d'éléments
40  * @param begin la position de début
   * @param end la position de fin
   */
43  public static void shiftRight(int[] is, int begin, int end){
       for (int i = end; i >= begin; i--){
           is[i+1] = is[i];
46  }
   }

```

java

## Supprimer

### Spécification

Supprimer une donnée d'un tableau de données triées.

**Données** : le tableau à modifier, le nombre d'éléments dans ce tableau, la valeur à supprimer

**Résultat** : le tableau reçu est modifié en lui supprimant la valeur

### Solution

Pour supprimer un élément, dès lors qu'il est trouvé, il suffit de décaler tous les éléments à sa droite vers la gauche et adapter la taille logique.

```
// supprimer l'élément donné
algorithm delete( is↓↑ : array of integer, nValues↓↑ : integer,
                                     value↓ : integer ) → integer
    integer index
    boolean isFound
    if verify(is, nValues, value) ≠ -1 then
        index = findIndex(is, nValues, value)
        shiftLeft(is, index + 1, nValues)
        nValues = nValues - 1

    return nValues

// Décale d'une position à gauche les éléments
// entre la position début et fin
algorithm shiftLeft( tab↓↑ : array of integers, begin↓ : integer, end↓ : integer )
    for i from begin to end
        is[i-1] = is[i]
```

*pseudocode*

```
public static int delete(int[] is, int nValues, int value){
    int index;
3   if (verify(is, nValues, value) != -1){
        index = findIndex(is, nValues, value);
        shiftLeft(is, index+1, nValues);
6       nValues = nValues - 1;
    }
    return nValues;
9 }

public static void shiftLeft(int[] is, int begin, int end){
12  for (int i=begin; i <= end; i++){
        is[i-1] = is[i];
    }
15 }
```

java





## n° 10

### Recherche dichotomique

#### Problème

Trouver rapidement la position d'une valeur donnée dans un tableau **trié** d'entiers. Si la valeur n'est pas présente, on donnera la position où elle aurait du se trouver.

Voir Section 11.3 page 135.

#### Spécification

**Données** : le tableau à analyser et la valeur recherchée

**Résultat** : un booléen indiquant si la valeur a été trouvée et un entier indiquant soit la position où la valeur a été trouvée soit la position où elle aurait du être.

#### Solution

L'algorithme rapide que nous avons vu est la recherche dichotomique.

```

public static int binarySearch(int[] myArray, int value){
2   int leftIndex = 0;
    int medianIndex = -1;
    int rightIndex = myArray.length - 1;
5   int candidate;
    boolean isFound = false;

8   while (!isFound && leftIndex <= rightIndex){
        medianIndex = (leftIndex + rightIndex) / 2;
        candidate = myArray[medianIndex];
11  if (candidate == value){
            isFound = true;
        } else if (value < candidate){
14      rightIndex = medianIndex - 1;
        } else {
            leftIndex = medianIndex + 1;
17  }
    }

20  if (isFound){
        return medianIndex;
    } else {
23      return -1;
    }
}

```

java

Dans le cas d'un langage permettant d'autres passages de paramètres que le passage par valeur (cfr. Section 7.3 page 81), nous pourrions écrire un algorithme :

- ▷ qui retourne un booléen précisant si la valeur est trouvée et ;
- ▷ qui donne la position à laquelle se trouve la valeur ou la position à laquelle elle devrait se trouver ce qui serait pratique en cas d'insertion.

Un tel algorithme aurait cette allure :

```
algorithm dichotomousSearch(
    myArray↓ : array of integers, value↓ : integer, pos↓↑ : integer ) → boolean
    integer rightIndex, leftIndex, medianIndex
    integer candidate
    boolean isFound

    leftIndex = 0
    rightIndex = myArray.length - 1
    isFound = false

    while NON isFound AND leftIndex ≤ rightIndex
        medianIndex = (leftIndex + rightIndex) DIV 2
        candidate = myArray[medianIndex]
        if candidate == value then
            isFound = vrai
        else if candidate < value then
            leftIndex = medianIndex + 1           // on garde la partie droite
        else
            rightIndex = medianIndex - 1         // on garde la partie gauche

    if isFound then
        pos = medianIndex
    else
        pos = leftIndex           // dans le cas où la valeur n'est pas trouvée,
        // on vérifiera que leftIndex donne la valeur où elle pourrait être insérée.

    return isFound
```

*pseudocode*



## n° 11

### Tri d'un tableau

Trier un tableau d'entiers par ordre croissant. Les algorithmes de tri sont présentés aux sections 12.2 à 12.4 pages 141–145.

### Spécification

**Données** : le tableau non trié, à trier.

**Résultat** : ce même tableau trié.

### Solution

#### Le tri par insertion

```
// Trie le tableau reçu en paramètre (via un tri par insertion).
algorithm insertionSort(myArray $\downarrow\uparrow$  : array of integers)
    integers i, j, value
    for i from 1 to myArray.length - 1
        value = myArray[i]
        // recherche de l'endroit où insérer value dans le
        // sous-tableau trié et décalage simultané des éléments
        j = i - 1
        while j  $\geq$  0 AND value < myArray[j]
            myArray[j+1] = myArray[j]
            j = j - 1
        myArray[j+1] = value
```

*pseudocode*

```

1 /**
   * Tri par insertion.
   *
4  * @param myArray tableau à trier
   */
   public static void insertionSort(int[] myArray) {
7       for (int nbSortedValues = 1;
           nbSortedValues < myArray.length;
           nbSortedValues++) {
10          int value = myArray[nbSortedValues];
           int pos = nbSortedValues - 1;
           while (pos >= 0 && value < myArray[pos]) {
13              myArray[pos + 1] = myArray[pos];
              pos = pos - 1;
           }
16          myArray[pos + 1] = value;
       }
   }

```

java

## Le tri par sélection des minima successifs

```

// Trie le tableau reçu en paramètre (via un tri par sélection des minima successifs).
algorithm selectionSort(myArray↓↑ : array of integers)
    integer i, minIndex
    for i from 0 to myArray.length - 2      // i correspond à l'étape      // de
l'algorithme
        minIndex = searchMinIndex( myArray, i, myArray.length - 1 )
        swap( myArray[i], myArray[minIndex] )

// Retourne l'indice du minimum entre les indices début et fin du tableau reçu.
algorithm searchMinIndex( myArray↓ : array of integers,
                           début↓, fin↓ : integers) → integer

    integers minIndex, i
    minIndex = début
    for i from début+1 to fin
        if myArray[i] < myArray[minIndex] then
            minIndex = i

    return minIndex

// Échange le contenu de 2 variables.
algorithm swap(a↓↑, b↓↑ : integers)
    integer aux
    aux = a
    a = b
    b = aux

```

pseudocode

```

/**
 * Trie le tableau reçu en paramètre via un tri par sélection
3  * des minima successifs
 *
 * @param myArray le tableau à trier
6  */
public static void selectionSort(int[] myArray) {
    for (int i = 0; i < myArray.length - 1; i++) {
9      int minIndex = searchMinIndex(myArray, i, myArray.length -
        1);
        if (minIndex != i) {
            swap(myArray, i, minIndex);
12     }
    }
}

15 /**
 * Retourne l'indice du minimum entre les indices début et fin
18 * du tableau reçu.
 *
 * En cas d'égalité, c'est l'indice le plus petit qui est retourné.
21 *
 * @param myArray le tableau d'entiers
 * @param begin l'indice de début
24 * @param end l'indice de fin
 */
public static int searchMinIndex(
27     int[] myArray,
        int begin,
        int end) {
30     int minIndex = begin;
    for (int i = begin + 1; i <= end; i++) {
        if (myArray[i] < myArray[minIndex]) {
33             minIndex = i;
        }
    }
36     return minIndex;
}

39 /**
 * Échange deux valeurs dans un tableau
 *
 * @param myArray un tableau
 * @param pos1 une position
 * @param pos2 une (autre) position
45 */
private static void swap(int[] myArray, int pos1, int pos2) {
    int value = myArray[pos1];
48     myArray[pos1] = myArray[pos2];
    myArray[pos2] = value;
}

```

java

## Le tri bulle

```
// Trie le tableau reçu en paramètre (via un tri bulle).
algorithm bubbleSort(myArray $\downarrow\uparrow$  : array of integers)
    integers burstIndex, bubbleIndex
    for burstIndex from 0 to myArray.length - 2
        for bubbleIndex from myArray.length - 2 to burstIndex by -1
            if myArray[bubbleIndex] > myArray[bubbleIndex + 1] then
                // voir algorithme précédent
                swap( myArray[bubbleIndex], myArray[bubbleIndex + 1] )
```

*pseudocode*

```
1 /**
   * Tri bulle d'un tableau d'entiers.
   *
4  * @param myArray
   */
   public static void bubbleSort(int[] myArray) {
7     // La bulle part de la droite,
     // en faisant remonter l'élément le plus petit,
     // et éclate à gauche.
10    for (int burstPos = 0;
        burstPos < myArray.length - 1;
        burstPos++) {
13        for (int bubblePos = myArray.length - 1;
            bubblePos > burstPos;
            bubblePos--) {
16            if (myArray[bubblePos] < myArray[bubblePos - 1]) {
                swap(myArray, bubblePos, bubblePos - 1);
            }
19        }
    }
22 }

/**
 * Échange deux valeurs dans un tableau
25 *
 * @param myArray un tableau
 * @param pos1 une position
28 * @param pos2 une (autre) position
 */
   private static void swap(int[] myArray, int pos1, int pos2) {
31     int value = myArray[pos1];
     myArray[pos1] = myArray[pos2];
     myArray[pos2] = value;
34 }
```

java

# Annexe D

## Bonnes pratiques

Ce chapitre recense quelques bonnes pratiques d'écriture d'algorithme et de code. Il montre également des manières d'écrire du code peu lisibles et donc à éviter, voire à proscrire.

Les bonnes pratiques d'écriture ont également un aspect culturel et dépendant d'un langage. En effet, certaines communautés de programmeurs habitués à un langage adoptent des styles de programmation qui ne sont pas partagés par d'autres programmeurs, habitués à un autre langage. Nous présentons ici des bonnes pratiques d'algorithmique générale et des bonnes pratiques liées au langage Java.

« Lorsque je code en langage Java, je respecte les conventions du langage Java. Lorsque je code en langage C++, je respecte les conventions du langage C++.

Lorsque je contribue à un projet, je respecte les conventions du langage et celles proposées par l'équipe en charge du projet. »

### Contenu

D.1	Bonnes pratiques générales . . . . .	<b>263</b>
D.2	Tester un booléen . . . . .	<b>264</b>
D.3	Assigner un booléen . . . . .	<b>265</b>
D.4	Assigner une valeur en fonction d'une condition . . . . .	<b>265</b>
D.5	Interrompre une boucle <b>for</b> . . . . .	<b>266</b>
D.6	Plusieurs <b>return</b> . . . . .	<b>267</b>

### D.1 Bonnes pratiques générales

Cette section ne fait pas partie de la matière évaluée dans la partie théorique.

Certaines conventions d'écriture, sont communes à tous les langages. Citons par exemple :

- ▷ préférer des lignes de longueur maximale de 80 caractères. Ceci facilite la lecture ;

- ▷ couper les lignes qui seront trop longues en suivant ces règles ;
  - couper après une virgule ;
  - couper avant un opérateur ;
  - préférer les coupures de haut niveau de priorité aux coupures de priorité faible ;
  - aligner la nouvelle ligne avec le début de l'expression se trouvant à la ligne précédente.

Nous vous invitons à consulter les conventions d'écriture du code en Java — même si le document est vieux — ceci afin d'écrire un code plus lisible. Voir [\[tea99\]](#)

## D.2 Tester un booléen

Lorsqu'il s'agit de comparer deux nombres dans la condition d'un **if**, nous écrivons assez naturellement : **if** (**nb1**<**nb2**) et il ne vous viendrait pas à l'idée d'écrire : **if** (**nb1**<**nb2** == **true**).

Avec un booléen, nous allons également éviter d'écrire == **true** ou == **false**. Par exemple, pour exécuter un traitement si la variable **isAdult** est vraie, certaines personnes programmeuses débutantes écrivent ceci :



```
if (isAdult == true){  
2  // ...  
}
```

java

Cette écriture est correcte mais inutilement lourde. La variable étant déjà un booléen qui vaut vrai ou faux, il suffit d'écrire :

```
if (isAdult){  
2  // ...  
}
```

java

C'est une bonne pratique d'utiliser cette deuxième écriture.

De même, si le test doit être vrai quand la variable booléenne est fausse, il suffit d'écrire :

```
if (!isAdult){  
2  // ...  
}
```

java



## D.3 Assigner un booléen

Pour assigner une valeur booléenne à une variable, il n'est pas utile d'utiliser un `if-else`, une assignation directe fait l'affaire.

Par exemple, si la variable booléenne `isNul` doit valoir `true` si un nombre est égal à 0 et `false` sinon, nous pourrions écrire le bout de code ci-dessous.

```
1 if (nb == 0){  
2   isNul = true;  
3 } else {  
4   isNul = false;  
5 }
```

java



Ce code est correct mais inutilement compliqué. Puisque on assigne `true` si la condition est vraie et `false` si la condition est fausse, il suffit d'assigner la condition. Ce qui donne :

```
isNul = nb == 0;
```

java

## D.4 Assigner une valeur en fonction d'une condition

Cette section ne fait pas partie de la matière évaluée dans la partie théorique.

Examinons l'exemple ci-contre qui assigne un prix `price` qui doit être différent en fonction du droit ou non à un tarif réduit.

```
1 if (isDiscount){  
   price = 8;  
2 } else {  
3   price = 12;  
4 }
```

java

Nous rencontrons parfois le code suivant. Plus court. Est-il nécessairement mieux ?

```
price = 12;  
if (isDiscount){  
3   price = 8;  
}
```

java



Cette manière d'écrire comporte effectivement (un peu) moins de lignes mais ce critère n'a que très peu d'importance. Cette version n'est pas plus rapide (elle est

même plus lente en cas de tarif réduit) et elle est moins lisible car le lecteur croit d'abord que le tarif est toujours de 12 avant de constater qu'il peut être différent.

Nous privilégions la première version.

## D.5 Interrompre une boucle `for`

Certains développeurs et développeuses débutantes<sup>81</sup> interrompent anticipativement une boucle `for` *en modifiant l'indice*. C'est une mauvaise idée.

Dans l'exemple ci-dessous, l'indice `i` reçoit une « grande valeur » pour sortir de la boucle `for`.



```
1 for (int i=0; i < n; i++){  
    // ...  
    if (aCondition) {  
4     i = n + 1;    // pour interrompre la boucle  
    }  
    // ...  
7 }
```

java

Cette manière de faire est à proscrire absolument et ce, quelle qu'en soit la raison. Ce n'est pas une bonne idée en terme de lisibilité et dans certains langages c'est interdit par le compilateur ou cela ne fait pas ce qui est attendu.

Si vous devez interrompre un parcours répétitif, lorsqu'une condition est remplie, nous insistons pour l'utilisation d'une boucle **while** et l'utilisation d'un booléen pour contrôler la fin.

Par exemple :

```
1 int i = 0;  
  isFinished = false;  
  while (i < n && !isFinished){  
4    // ...  
    if (aCondition){  
        isFinished = true;  
7    }  
    // ...  
    i = i + 1;  
10 }
```

java

Si le test est la première instruction de la boucle et que la condition est courte, on peut aussi se passer de la variable booléenne.

<sup>81</sup>. En essayant d'avoir une écriture plus inclusive, je m'autorise un accord de proximité.

```
1 int i = 0;
  while (i < n && !aCondition){
    // ...
4   i = i + 1;
  }
```

java

Si vous tenez absolument à interrompre la boucle **for**, vous pouvez utiliser un **return** comme expliqué au point suivant.

## D.6 Plusieurs **return**

Certaines personnes ont ont comme règle :

« Un seul **return** par algorithme ».

Cette règle n'est pas partagée par tous les développeurs et développeuses. Il existe un courant qui tolère plusieurs **return** dans certains cas précis.

En voici quelques uns.

Un cas concret est l'arrêt anticipé d'une méthode ou d'une boucle. Certains écrivent :

```
for (int i = 0; i < n; i++){
  if (aCondition){
3   return something;
  }
  // ...
6   return somethingElse,
}
```

java

Les personnes qui préfèrent un seul **return** opteront systématiquement pour une boucle **while**.

Une autre situation est lorsque la valeur à retourner dépend d'un test. On voit alors des bouts de code qui ressemblent à :

```
1      if (aCondition){
        return something;
    } else {
4      return somethingElse;
    }
```

java

qui est acceptable mais peut-être avantageusement remplacé par l'opérateur ternaire :

```
return aCondition ? something : somethingElse;
```

java

Dans le cas où il y a un cas "simple" et un cas "complexe", on préférera traiter le cas simple d'abord :

```
1      if (aCondition){
2          return somethingSimple;
3      }
4
5      // more complex logic here
6
7      return result;
```

java

Enfin, si l'alternative est entre deux cas complexes, il faut absolument éviter les "gros if" qui sont un signe que notre code fait trop de choses. Ceci est donc mauvais :

```
1      if (aCondition){
2
3          // complex logic here
4
5          return something;
6      }
7
8      // also complex logic here
9
10     return somethingElse;
```

java

Dans ces cas, on préférera repenser son code et éventuellement introduire des méthodes, par exemple :

```
1      if (aCondition){
2          return methodCall();
3      }
4      return anotherMethodCall();
```

java

# Index

- affectation interne, 38
- afficher, 54
- algorithme, 15
- alternatives, 65
- aléatoire, 46
- AND, 42
- API, 166, 182
- api, 168
- appel, 53
- application programming interface, 166
- assignment, 38
  
- binaire, 41
- bloc, 36, 66
- boucles, 90
- break, 76
- byte, 152
  
- caractère, 166
- chaîne, 165
- chaîne (longueur d'une), 166
- classe, 53
- commentaires, 51
- comparaisons, 41
- comparators, 75
- compilation, 59
- concaténation, 167
- conditional expression, 132
- conditionals operators, 75
- constantes, 52
- court-circuit, 44
- création, 118
  
- declaration, 183
- demandeur, 54
- DIV, 45
- division entière, 45
- do-while, 96
- débogueur, 39
- déclaration, 36, 118
  
- décomposer le code, 79
  
- efficacité, 48
- entête, 33
- environnement, 61
- ET, 42
- exceptions, 175
- expression, 38, 41
- expression booléenne, 74
  
- fonction, 80
- for, 92
- foreach, 123, 239, 242
  
- grammaire, 63
  
- hasard, 46
  
- ide, 61
- if, 65
- if-else, 69
- implémenter un algorithme, 34
- indentation, 49
- initialisation, 118
- int, 85, 152
- interprétation, 59
  
- javadoc, 51, 181
- JDK, 62
- jls, 63
- JRE, 62
- junit, 108
- jvm, 60
  
- lisibilité, 48, 49
- litterate programming, 182
- littéral, 41, 151, 166
- long, 152
  
- majeur, 140
- mineur, 140

- MOD, 45
- module, 80, 81
- modulo, 45
- Morgan, 43
- mots-clés, 49
- méthode, 80
  
- noms, 24
- NON, 42
  
- opérande, 41
- opérateur, 41
- opérateur ternaire, 132, 237
- opérateurs d'égalité, 75
- opérateurs logiques, 42, 75
- opérateurs relationnels, 74
- OR, 42
- OU, 42
  
- paramètres, 81
- paramètres effectifs, 86
- paramètres formels, 86
- paramètres Java, 86
- plan de test, 107
- point d'entrée, 34
- portée, 36
- pour, 92
- primitif, 79
- priorité, 41, 43
- procédure, 80
- programme, 16
  
- rapidité, 48
- read, 54
- recherche dichotomique, 135
- reference, 79
- return, 33
- référence, 117
  
- scope, 36
- selon-que, 75
- short, 152
- si, 65
- si-sinon, 69
- si-sinon-si, 72
- signature, 183
- suite récurrent, 104
- switch, 75
- séquence d'échappement, 156
  
- tableau, 116, 117, 161
- tableau (indice), 117
- tableau (taille), 117
- tables de vérité, 42
- tag, 185
- taille logique, 123
- taille physique, 123
- tant que, 90
- ternaire, 41, 132
- test unitaire, 106
- tracer, 39
- tri, 139
- type primitif, 84, 151
- type référence, 84, 160
- type référence, 151
- types, 25
- types numériques, 84
  
- unaire, 41
  
- valeur sentinelle, 101
- variable, 24, 36, 85
- variable globale, 36
- variable locale, 36
- vérifier un algorithme, 32
  
- while, 90

# Références

- [Bet09] Pierre Bettens. Unicode, utf8, utf16, utf32, ... et tutti quanti. <http://blog.namok.be/?post/2009/11/30/unicode-UTF8-UTF16-UTF32-et-tutti-quanti>, novembre 2009.
- [Bet19] Pierre Bettens. Jdk10, 11 et 12, quelques nouveautés. <http://blog.namok.be/?post/2019/04/06/JDK10-11-12>, avril 2019.
- [GJS<sup>+</sup>17] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Language Specification. Java SE 9 Edition*. Oracle America Inc., 2017.
- [Sma90a] David Small. Hérésie. deuxième partie. [http://valvassori.free.fr/dave\\_small/heresie2.php3](http://valvassori.free.fr/dave_small/heresie2.php3), ±1990.
- [Sma90b] David Small. Hérésie. première partie. [http://valvassori.free.fr/dave\\_small/heresie1.php3](http://valvassori.free.fr/dave_small/heresie1.php3), ±1990.
- [tea99] Java team. Code conventions for the java tm programming language. <https://www.oracle.com/technetwork/java/codeconvtoc-136057.html>, 20 avril 1999.
- [tea18] Java team. Java® platform, standard edition & java development kit version 11 api specification. <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>, 2018. essai note.
- [VE04] Marion Videau and David Eck. Les algorithmes de tri. [http://interstices.info/jcms/c\\_6973/les-algorithmes-de-tri](http://interstices.info/jcms/c_6973/les-algorithmes-de-tri), 2004.
- [Wel00] Don Wells. Code the unit test first. <http://www.extremeprogramming.org/rules/testfirst.html>, 2000.

