

# Développement I (Algorithmique et Java)

DEV 1 - DEV

2018-2019

A.HALLAL  
C.LEIGNEL  
DP.BISCHOP  
J.BELEHO  
M.CODUTTI  
N.PETTIAUX  
N.RICHARD  
P.BETTENS



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

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


Document produit avec L<sup>A</sup>T<sub>E</sub>X.  
Version du 12 juin 2018.

## Crédits

 Icône « Book dictionary » 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 « Card » de Alexander Skowalsky<sup>5</sup> de The noun project

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

 Icône « Coffee » de Luis Prado<sup>7</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/sandorsz>
  6. <https://thenounproject.com/grega.cresnar>
  7. <https://thenounproject.com/Luis>

# Table des matières



## 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>8</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>7</b>
1.1.1	Préliminaires : utilité de l’ordinateur . . . . .	7
1.1.2	Poser le problème . . . . .	8
1.2	Procédure de résolution . . . . .	<b>9</b>
1.2.1	Chronologie des opérations . . . . .	9
1.2.2	Les opérations élémentaires . . . . .	9
1.2.3	Les opérations bien définies . . . . .	10
1.2.4	Opérations soumises à une condition . . . . .	11
1.2.5	Opérations à répéter . . . . .	11
1.2.6	À propos des données . . . . .	11
1.3	Ressources . . . . .	<b>12</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>8</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 : « Calculer la moyenne de nombres entiers. ».

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>9</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 rond » 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).

9. 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.

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 à l'aide du four à micro-ondes.

#### 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 perpétuellement.

#### 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.

### 1.3 Ressources

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

- ▷ Les Sépas 18 - Les algorithmes<sup>10</sup>
- ▷ Les Sépas 11 - Un bug<sup>11</sup>
- ▷ Le crépier psycho-rigide comme algorithme<sup>12</sup>
- ▷ Le baseball multicolore comme algorithme<sup>13</sup>
- ▷ Le jeu de Nim comme algorithme<sup>14</sup>

---

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

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

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

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

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

## Chapitre 2

# Une approche ludique : Code Studio



Il existe de nombreux programmes qui permettent de s'initier à la création d'algorithmes. Nous voudrions mettre en avant le projet *Code Studio*. Soutenu par de grands noms de l'informatique comme Google, Microsoft, Facebook et Twitter, il permet de s'initier aux concepts de base au travers d'exercices ludiques faisant intervenir des personnages issus de jeux que les jeunes connaissent bien comme Angry birds ou Plantes et zombies.

Sur le site Code Studio<sup>15</sup> nous avons sélectionné pour vous :

- ▷ **L'heure de code** : <http://studio.code.org/hoc/1>.  
Un survol des notions fondamentales en une heure au travers de vidéos explicatives et d'exercices interactifs.
- ▷ **Cours d'introduction** : <http://studio.code.org/s/20-hour>.  
Un cours de 20 heures destiné aux adolescents. Il reprend et approfondit les éléments effleurés dans « L'heure de code »

Nous vous conseillons de créer un compte sur le site ainsi vous pourrez retenir votre progression et reprendre rapidement votre travail là où vous l'avez interrompu.

Votre professeur pourra vous guider dans votre apprentissage.

---

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



# Chapitre 3

## 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

3.1	Algorithmes et programmes . . . . .	<b>15</b>
3.1.1	Algorithme . . . . .	15
3.1.2	Programme . . . . .	16
3.1.3	Les constituants principaux de l'ordinateur . . . . .	17
3.1.4	Exécution d'un programme . . . . .	18
3.2	Les phases d'élaboration d'un programme . . . . .	<b>18</b>
3.3	Conclusion . . . . .	<b>20</b>
3.4	Ressources . . . . .	<b>20</b>

### 3.1 Algorithmes et programmes

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

#### 3.1.1 Algorithme

Un algorithme appartient au vaste ensemble des *marches à suivre*.

**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 — à priori le langage naturel — mais il y a d'autres possibilités : ordino-gramme, arbre programmatique, pseudo-code. . .

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

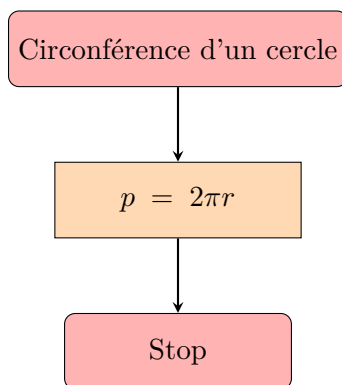
### Exemple simple

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

$$p = 2\pi r$$

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
  type var
  return 2 π radius
  
```

*pseudocode*

### 3.1.2 Programme



Un **programme** n'est rien d'autre que la représentation d'un algorithme dans un langage plus technique compris par un ordinateur (par exemple : Assembleur, Cobol, Java, C++...). 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

### 3.1.3 Les constituants principaux de l'ordinateur

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

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

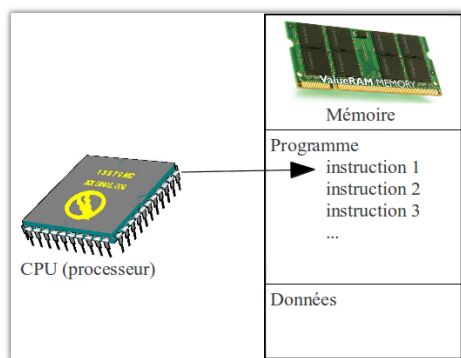
- ▷ **l'organe de contrôle** : c'est le cerveau de l'ordinateur. Il est l'organisateur, le contrôleur suprême de l'ensemble. Il assume l'enchaînement des opérations élémentaires. Il s'occupe également d'organiser l'exécution effective de ces opérations élémentaires reprises dans les programmes.
- ▷ **l'organe 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 ou la fréquence d'utilisation ne justifient pas la conservation permanente en mémoire centrale.

Le **software 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).

### 3.1.4 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 ?

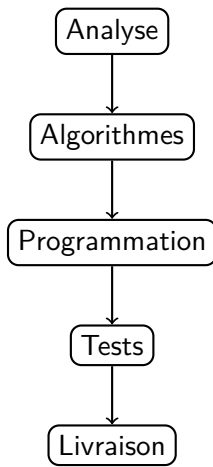
De façon très simplifiée, les étapes suivantes ont lieu :

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 !

## 3.2 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 dans le cours d'analyse.
- ▷ Une fois le problème analysé, et avant de passer à la phase de programmation, il faut réfléchir à **l'algorithme** qui va permettre de résoudre le problème.
- ▷ On peut alors **programmer** cet algorithme dans le langage de programmation choisi ; Java, Cobol, Assembleur, 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.

### 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 ? Le langage que nous allons utiliser en algorithmique est plus souple et plus général que le langage Java par exemple (où il faut être précis au « ; » près).
- ▷ 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 ? ». L'algorithme est plus adapté à la communication car plus lisible.
- ▷ 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.

### 3.3 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.

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.

### 3.4 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>16</sup>

---

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

## Deuxième partie

# Les bases de l'algorithmique et de la programmation



Chapitre

4

# 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

4.1	Déterminer les données et le résultat . . . . .	23
4.2	Les noms . . . . .	24
4.3	Les types . . . . .	25
4.3.1	Il n’y a pas d’unité . . . . .	26
4.3.2	Préciser les valeurs possibles . . . . .	26
4.3.3	Le type des données complexes . . . . .	27
4.3.4	Exercice . . . . .	27
4.4	Résumés . . . . .	27
4.4.1	Résumé graphique . . . . .	27
4.4.2	Résumé textuel . . . . .	28
4.5	Exemples numériques . . . . .	28

## 4.1 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>17</sup>. Dans la suite, nous utiliserons un exemple très simple afin d’illustrer les concepts qui nous intéressent.

17. 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.

## 4.2 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>18</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 changer — 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>18</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 ou des méthodes lorsque l'on implémentera ces algorithmes.

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 5.5, 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.

## 4.3 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 ces **types**<sup>19</sup> :

Algo	Java	
integer	int	pour les nombres entiers
real	double	pour les nombres réels
string	String <sup>20</sup>	pour les chaînes de caractères, les textes par exemple : "Bonjour", "Bonjour le monde!", "a", "..."
boolean	boolean	quand la valeur ne peut être que <code>true</code> (vrai) ou <code>false</code> (faux)

### Exemples.

- ▷ Pour la longueur, la largeur et la surface d'un rectangle, on prendra un réel.
- ▷ Pour le nom d'une personne, on choisira la chaîne.

<sup>19</sup> Écrire ces différents types en français n'est pas une faute. Dans ces notes nous utiliserons l'anglais

- ▷ Pour l'âge d'une personne, un entier est indiqué.
- ▷ Pour décrire si un étudiant est doubleur ou pas, un booléen 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.

### 4.3.1 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`.

### 4.3.2 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.

### 4.3.3 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 loin dans le cours, nous verrons qu'il est possible de définir de nouveaux types de données grâce aux *structures* pour les algorithmes et aux *classes* pour les langages orientés objets tels que Java.

Nous pourrions alors 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.

### 4.3.4 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 ?

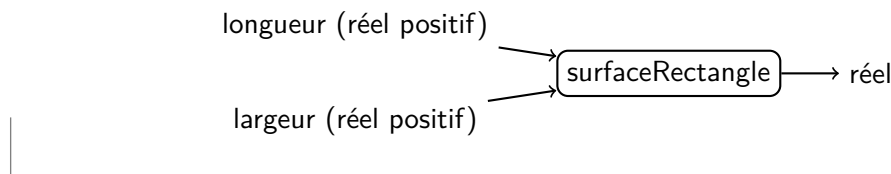
## 4.4 Résumés

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

### 4.4.1 Résumé graphique

Représentation graphique du problème.

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



#### 4.4.2 Résumé textuel

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

**Exemple.**

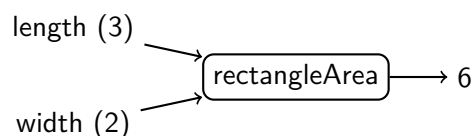
<b>Données</b>	longueur (un réel positif) largeur (un réel positif)
<b>Résultat</b>	la surface du rectangle

### 4.5 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 5

## 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 langue français, en pseudo-code ou avec un organigramme.

## Contenu

---

5.1	Exercice résolu : un problème simple . . . . .	<b>31</b>
5.1.1	Trouver l'algorithme . . . . .	31
5.1.2	Vérifier l'algorithme . . . . .	33
5.1.3	Implémenter l'algorithme en Java . . . . .	33
5.1.4	Résolution d'un second problème simple . . . . .	34
5.2	Décomposer les calculs ; variables et assignation . . . . .	<b>35</b>
5.2.1	Les variables . . . . .	35
5.2.2	L'assignation . . . . .	37
5.2.3	Tracer un algorithme . . . . .	38
5.2.4	Exercice résolu : durée du trajet . . . . .	39
5.3	Quelques difficultés liées au calcul . . . . .	<b>39</b>
5.3.1	Un peu de vocabulaire . . . . .	40
5.3.2	Les comparaisons et les assignations de variables booléennes . . . . .	41
5.3.3	Les opérations logiques . . . . .	41
5.3.4	La division entière et le reste . . . . .	44
5.3.5	Le hasard et les nombres aléatoires . . . . .	45
5.4	Des algorithmes et des programmes de qualité . . . . .	<b>46</b>
5.4.1	L'efficacité . . . . .	47
5.4.2	La lisibilité . . . . .	47
5.4.3	La rapidité . . . . .	48
5.4.4	La taille . . . . .	48
5.4.5	Conclusion . . . . .	48
5.5	Améliorer la lisibilité . . . . .	<b>49</b>
5.5.1	Mise en page des algorithmes . . . . .	49
5.5.2	Écriture des programmes . . . . .	50
5.5.3	Choix des noms . . . . .	50
5.5.4	Les commentaires . . . . .	50
5.5.5	Constantes . . . . .	52
5.6	Appel d'algorithme, appel de méthode . . . . .	<b>53</b>
5.7	Interagir avec l'utilisateur . . . . .	<b>54</b>
5.7.1	Afficher un résultat . . . . .	54
5.7.2	Demander des valeurs . . . . .	54
5.7.3	Préférer les paramètres . . . . .	55

---

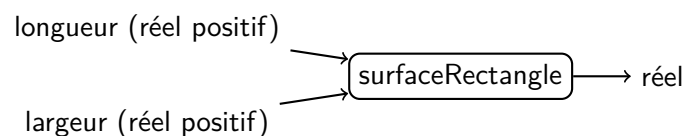
## 5.1 Exercice résolu : un problème simple

### 5.1.1 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.**

▷ `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

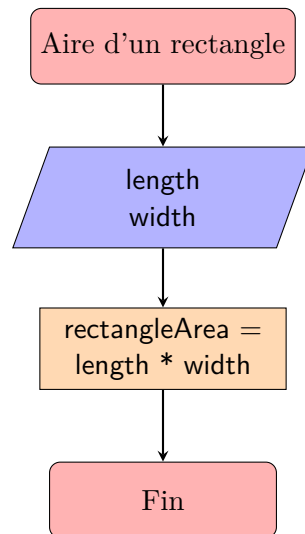
**Comment résoudre ce problème ?** La toute première étape est de comprendre le lien entre les données et le résultat. Ici, on va se baser sur la formule de la surface :

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

La surface s'obtient donc en multipliant la longueur par la largeur <sup>21</sup>.

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

21. 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 essaierons de nous concentrer sur des problèmes simples voire très simples.

ORGANIGRAMME 2 – Solution de `rectangleArea`

Un **pseudo-code** d'une solution s'écrit :

```

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 `_` font l'affaire.



### 5.1.2 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	✓

### 5.1.3 Implémenter l'algorithme en Java

Maintenant que l'algorithme est écrit, il reste à l'implémenter en 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

**Attention :**

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

- ▷ spécifier le problème ;
- ▷ fournir des exemples ;
- ▷ écrire un algorithme ;
- ▷ vérifier les exemples ;
- ▷ écrire un programme correspondant à l'algorithme ;
- ▷ tester le programme et constater qu'il fournit bien les résultats attendus

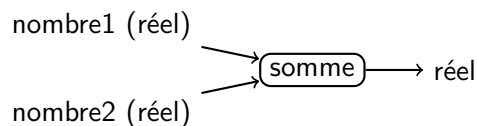
#### 5.1.4 Résolution d'un second problème simple



Vous pouvez vous baser sur la fiche ?? page ?? 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 :

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

**Algorithme** L'algorithme s'écrit simplement :

```
algorithm addition(number1, number2 : reals) → real
    return number1 + number2
```

pseudocode

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	✓

**Programme** 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

## 5.2 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.

### 5.2.1 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. 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 connue de toutes les classes et les méthodes d'un même projet. Nous y reviendrons.

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.

*pseudocode*

Mais, bien entendu, cette formulation, trop proche du langage parlé, serait trop floue et trop longue. Dès lors, nous abrègerons par :

réels length, width

*pseudocode*

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.

### 5.2.2 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.

En algorithmique, nous écrirons :


```
variableName = expression
```

*pseudocode*

En Java, nous utiliserons exactement le même symbole = :

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

java

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 // pour une variable logique
aString = "hello"
aString = hello // quelle différence avec la précédente ?
```

*pseudocode*

Et d'autres qui ne le sont pas :

```
sum + 1 = 3 // sum + 1 n'est pas une variable
sum = 3n // 3n n'est ni un nom de variable correct ni une expression correcte
```



*pseudocode*

Et ces même exemples — corrects — en Java :

```
1 denRes = den1 * den2;
  count = count + 1;
  average = (number1 + number2) / 2;
4 isALowerthanB = a < b;
  aString = "hello";
  aString = hello;
7
```

java

### Remarques

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

Ainsi, l'assignation  $\text{cpt} = \text{cpt} + 1$  ne veut pas dire que  $\text{cpt}$  et  $\text{cpt} + 1$  sont égaux, ce qui est mathématiquement faux mais que la *nouvelle* valeur de  $\text{cpt}$  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.

### 5.2.3 Tracer un algorithme

Pour vérifier qu'un algorithme est correct, le développeur ou la développeuse sera souvent amené·e 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.

```

1: integer a, b, c
2: a = 12
3: b = 5
4: c = a - b
5: a = a + c
6: b = a

```

pseudocode

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

1: integer a, b, c  
 2: a = 12  
 3: c = a - b  
 4: d = c - 2

*pseudocode*

#	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 !

#### 5.2.4 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 5.5 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 ?? page ?? qui présente cette solution.

### 5.3 Quelques difficultés liées au calcul

Vous êtes habitués à effectuer des calculs. L'expérience nous montre toutefois que certains calculs vous posent des difficultés. Soit parce que ce sont des opérations que vous utilisez peu, soit parce que vous n'avez pas l'habitude de les voir comme des 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.

### 5.3.1 Un peu de vocabulaire

La première difficulté que rencontre un ou une apprenant·e est généralement liée au vocabulaire utilisé. 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  $*$ .

#### 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).

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



### 5.3.2 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

```
boolean positive, adult, successful, perfect
positive = nb > 0           // positive est mis à vrai si le nb est positive
adult = age ≥ 21           // adult est vrai si l'âge est 21 ou plus
successful = cote ≥ 10     // successful est mis à vrai si la cote est //
supérieure ou égale à 10
perfect = errors == 0      // c'est perfect si le nombre de fautes est 0
```

*pseudocode*

En Java

```
boolean positive,
2  adult,
   successful,
   perfect;
5 positive = nb > 0;
  adult = age >= 21;
  successful = code >= 10;
8 perfect = errors == 0;
```

java

**Remarque** Pour tester l'égalité, nous utilisons le symbole `==` afin de le différencier du symbole `=` qui est le symbole de d'assignation. C'est la même chose en Java et dans beaucoup d'autres langages.

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

- ▷ La variable booléenne *néгатif* 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.

### 5.3.3 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 ET b
V	V	V
V	F	F
F	V	F
F	F	F

a	b	a OU b
V	V	V
V	F	V
F	V	V
F	F	F

a	NON a
V	F
F	V

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>22</sup> et le OR (OU), `||`<sup>23</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>24</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 de d'une disjonction de deux propositions est la conjonction des deux négations. »

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

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

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

... et s'écrit plus simplement :

$$\text{NON } (a \text{ ET } b) \Leftrightarrow \text{NON } a \text{ OU NON } b$$

$$\text{NON } (a \text{ OU } b) \Leftrightarrow \text{NON } a \text{ ET NON } b$$

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 parenthèse pour aider à la 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 la 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.**

```
ok = 1/b < 0.1      // provoque une erreur et un arrêt de l'algorithme si b=0.
ok = b≠0 ET 1/b < 0.1 // donne la valeur faux à ok si b=0 (court-circuit).
ok = 1/b < 0.1 ET b≠0 // provoque une erreur et un arrêt si b=0.
```

*pseudocode*

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

### 5.3.4 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  $r$

Dans nos algorithmes, nous noterons la division entière **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 |

#### Utilité - Tester la divisibilité

Les deux opérateurs **MOD** et **DIV**, respectivement `%` et `/`, permettent de tester si un nombre est un multiple d'un autre.

Si l'on veut par exemple 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.

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

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

Et, en langage java :

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

java

### 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 :**  $(65536 \text{ DIV } 100) \text{ MOD } 10 = 5$ .

#### 5.3.5 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>25</sup>. À partir

<sup>25</sup>. Pour être précis, nous devrions dire pseudo-hasard ou algorithmes et programmes pseudo-aléatoires.

d'un nombre donné<sup>26</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>27</sup>

```
int 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)

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

▷ un entier entre 0 inclus et n exclu

```
algorithm random(n : integer) → integer
|   return random() * n           // le nombre réel est tronqué
```

pseudocode

▷ un entier entre min et max inclus

```
algorithm random(min : integer, max : integer) → integer
|   return random(max-min+1) + min
```

pseudocode

## 5.4 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 :

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

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

**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.

#### 5.4.1 L'efficacité

L'**efficacité** désigne le fait que l'algorithme (le programme) résout<sup>28</sup> bien le problème donné. C'est un minimum !

#### 5.4.2 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>29</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 5.5 page 49 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.

---

28. À ne pas confondre avec *l'efficience* qui indique qu'il est économe en ressources.

29. On parle du processus de *déverminage* (ou *debugging* en anglais).

### 5.4.3 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'*efficience* qui traite de la gestion économe des ressources. Nous reparlerons de rapidité dans le chapitre consacré à la *complexité* des algorithmes.

Le langage Java est réputé lent. C'est une affirmation péremptoire et peut-être périmée. Bien sûr, comme le *bytecode* est interprété, 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.

### 5.4.4 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.

### 5.4.5 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>30</sup>.

---

30. 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.



## 5.5 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 ?

### 5.5.1 Mise en page des algorithmes

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>31</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.
- ▷ Des lignes verticales relient le début et la fin de quelque chose. Ici, un algorithme mais on pourra l'utiliser également pour les choix et les boucles.

Exemples à ne pas suivre.

```
algorithm duréeTrajet(vitesseMS, distanceKM : réels) → réel
réel distanceM
distanceM = 1000 * distanceKM
return distanceM / vitesseMS
```



pseudocode

```
algorithm duréeTrajet(vitesseMS, distanceKM : réels) → réel
  réel distanceM
  distanceM = 1000 * distanceKM    retourner distanceM / vitesseMS
```



pseudocode

Il faudra préférer

```
algorithm duréeTrajet(vitesseMS, distanceKM : réels) → réel
  réel distanceM
  distanceM = 1000 * distanceKM
  return distanceM / vitesseMS
```

pseudocode

31. 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.

### 5.5.2 Écriture des programmes

Pour l'écriture d'un programme, les règles sont semblables. L'utilisation d'un éditeur de code<sup>32</sup> ou d'un IDE (Environnement de Développement Intégré)<sup>33</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>34</sup>.
- ▷ S'il est nécessaire de couper une ligne trop longue, celle-ci est coupée avant un opérateur ou après une virgule.

```
1 public static boolean iDoNothing(double beautifulDouble,  
    int firstInteger, boolean isItReallyTrue){  
    double notSoBeautiful = beautifulDouble * firstInteger;  
4    return notSoBeautiful > 100 && notSoBeautiful < 10000  
        || isItReallyTrue;  
    }  
7
```

java

### 5.5.3 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.

### 5.5.4 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 :

32. 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++*

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

34. 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.

- ▷ 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.

Voici quelques mauvais commentaires

```
// Exemples de mauvais commentaires
real length
sum = 0
```

```
// La longueur est un réel .
// On initialise la somme à 0
```



*pseudocode*

```
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, l'instruction

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

*pseudocode*

dépourvue de commentaires est bien préférable aux lignes suivantes :

```
c1 = c0 * (1 + r / 100)           // calcul du nouveau capital
// c1 est le nouveau capital, c0 est l'ancien capital, t est le taux
```



*pseudocode*

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.

```
// Calcule la surface d'un rectangle dont on donne la largeur et la longueur.
// On considère que les données ne sont pas négatives.
algorithm rectangleArea(length, width : reals) → real
|   return length * width
```

*pseudocode*

```
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**

### 5.5.5 Constantes

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

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

Il est inutile de spécifier leur type — mais c'est autorisé — celui-ci étant défini implicitement par la valeur de la constante. 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**

```

constante PI = 3,1415
constante SEUIL_RÉUSSITE = 10
constante ESI = "École Supérieure d'Informatique"

```

pseudocode

```

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.

## 5.6 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 ne sont pas écrits sur une feuille mais rassemblés dans un ou plusieurs fichiers. Dans un premier temps, nous écrirons nos programmes dans un seul fichier. 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`

Pour pouvoir appeler une méthode en Java, il faudra qu'elle existe dans la même classe<sup>35</sup>.

L'appel d'un algorithme s'écrit ainsi :

<sup>35</sup>. Nous relâcherons assez vite cette contrainte

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

```
// On appelle l'algorithme
```

*pseudocode*

L'appel d'une méthode s'écrit ainsi :

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

java

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

## 5.7 Interagir avec l'utilisateur

### 5.7.1 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 rectangleArea(122, 3.78)
```

*pseudocode*

L'instruction **print** signifie que l'algorithme doit, à cet endroit de l'algorithme 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 affichages et les demandes peuvent aussi se faire à l'aide d'un organigramme et d'un parallélogramme.

Afficher rectangleArea(122, 3.78)

### 5.7.2 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)

```

*pseudocode*

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>36</sup>.

```

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

```

**java**

### 5.7.3 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 :

```

// Calcule la surface d'un rectangle dont on donne la largeur et la longueur.
// On considère que les données ne sont pas négatives.
algorithm rectangleArea(length, width : reals) → real
  return length* width

algorithm AreaTest()
  reals length, width
  read length, width
  print rectangleArea(length, width)

```

*pseudocode*

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

Et voici une traduction en langage Java :

```
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);
```

java



# Chapitre 6

## 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

6.1	Introduction . . . . .	<b>57</b>
6.1.1	Compilation - interprétation . . . . .	58
6.2	Environnement de développement . . . . .	<b>61</b>
6.2.1	La base . . . . .	61
6.2.2	Écrire le code . . . . .	61
6.2.3	Exécuter le programme . . . . .	62
6.2.4	Hello world . . . . .	62
6.3	La grammaire du langage . . . . .	<b>63</b>

### 6.1 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.

La 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>37</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.

### 6.1.1 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.

**Définition**

---

<sup>37</sup>. Vous noterez la traduction.

Un langage est **compilé** (voir fig. 6.1) si le code source est traduit d'une traite. Un nouveau fichier contenant le code exécutable est créé.

Ces langages nécessitent un **compilateur**.

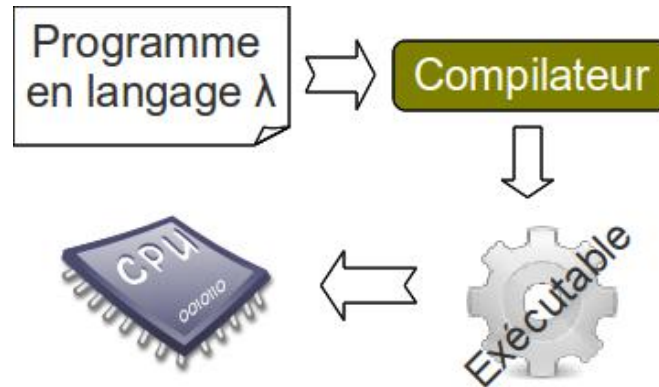


FIGURE 6.1 – Langage compilé

### Définition

Un langage est **interprété** (voir fig. 6.2) 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>38</sup>.

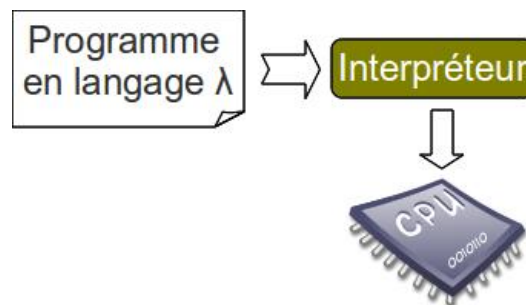


FIGURE 6.2 – Langage interprété

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.

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

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

- ▷ 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*).

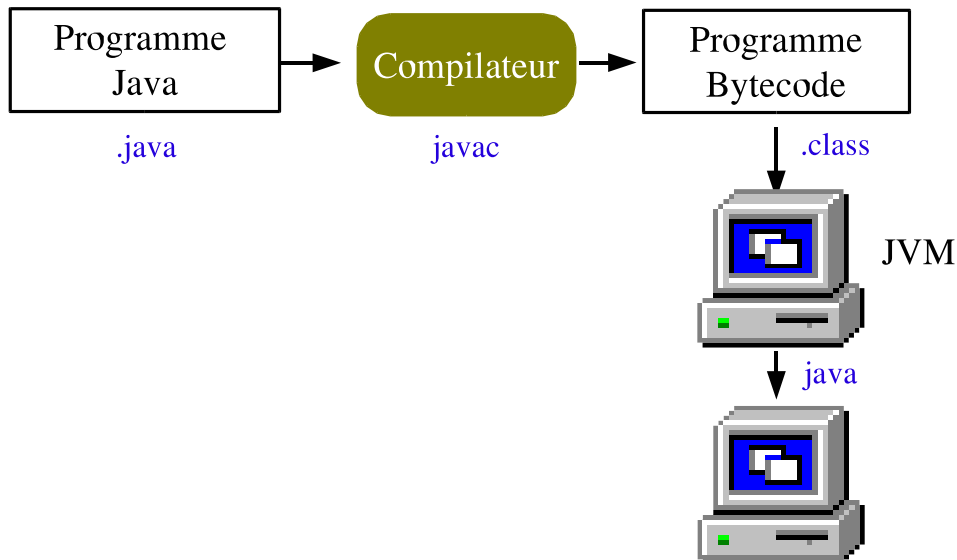


FIGURE 6.3 – Java est compilé et ensuite interprété

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 l'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** 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).

## 6.2 Environnement de développement

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

### 6.2.1 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.

### 6.2.2 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 inutile 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*...

Connaitre et utiliser correctement un éditeur de code et un IDE sont deux compétences essentielles d'un bon développeur.

### 6.2.3 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>39</sup>
2. OpenJDK est une alternative libre au JDK officiel Java.  
Télécharger OpenJDK <sup>40</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.

### 6.2.4 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**.

Grâce à mon éditeur de code, je crée le fichier `Hello.java` <sup>41</sup> qui contient :

```
1 public class Hello{  
    public static void main (String[] args) {  
        System.out.println("Hello_world");  
4    }  
}
```

java

Je compile ma classe :

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

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

41. Si vous êtes un utilisateur MS Windows, désactivez la propriété «*Hide extension for know 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`.

```
$  
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**.

## 6.3 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>42</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.

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
```

---

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

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>43</sup> p. 42.

---

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



# Chapitre 7

## Une question de choix

Dans la section 2 (p. 13) nous proposons une initiation ludique aux algorithmes. Vous avez eu l'occasion d'y aborder les alternatives. Par exemple, vous avez indiqué au zombie quelque chose comme : « S'il existe un chemin à gauche alors tourner à gauche ».

Les **alternatives** permettent de n'exécuter des instructions que si une certaine *condition* est vérifiée. Avec le zombie, par exemple, vous testiez son 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.

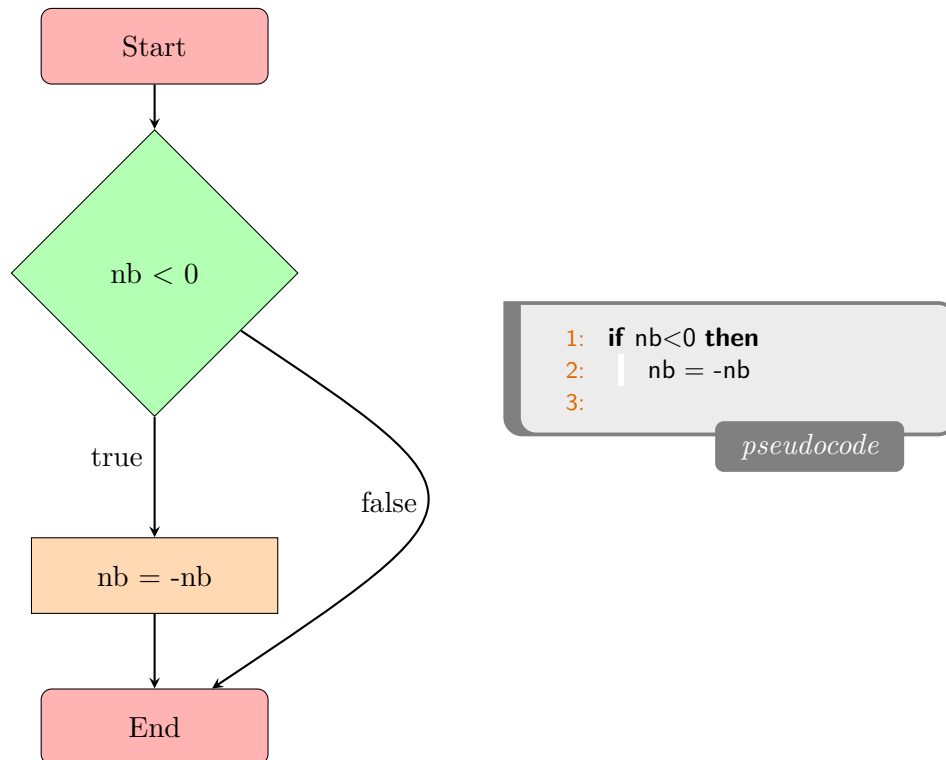
### Contenu

7.1	Le si ( <i>if-then</i> ) . . . . .	65
7.2	Le si-sinon ( <i>if-then-else</i> ) . . . . .	68
7.3	Le si-sinon-si . . . . .	70
7.4	Expression booléenne . . . . .	73
7.5	Le selon-que ( <i>switch</i> ) . . . . .	75

### 7.1 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.

**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, graphiquement et en pseudocode, un algorithme :



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

#	nb	test
1	-3	vrai
2	3	

#	nb	test
1	3	faux

La condition peut être n'importe quelle expression (calcul) dont le résultat est un booléen (vrai ou faux).



**Remarque** Attention, la confusion est fréquente.

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.

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

```
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. Un bloc d'instructions est toujours délimités par une paire d'accolades.

Le test précédent pourrait s'écrire comme suit :

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

java

Pour s'éviter des erreurs, nous utiliserons toujours le bloc d'instructions et nous écrirons :

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

java

**Exercice de compréhension** Tracez l'algorithme ou le programme avec les valeurs fournies et donnez la valeur de retour.

```
algorithm exercice(a, b : entiers) → entier
    entier c
    c = 2 * a
    if c > b then
        c = c - b
    return c
```

pseudocode

```
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

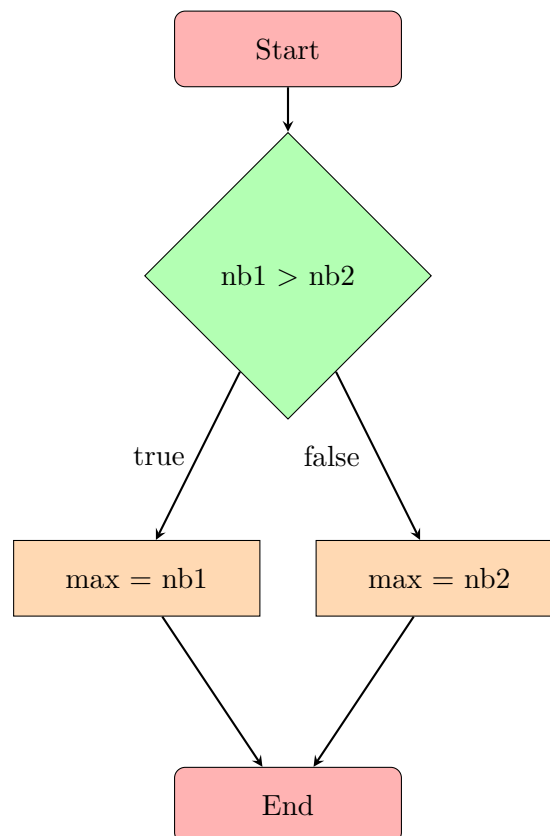
▷ exercice(2, 5) = \_\_\_\_\_

▷ exercice(4, 1) = \_\_\_\_\_

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

La construction si-sinon permet d'exécuter certaines instructions ou d'autres en fonction d'un test. Pour illustrer cette instruction, nous allons nous pencher sur un grand classique, la recherche de maximum.

**Exemple.** Pour déterminer le 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.



```
1: if nb1>nb2 then
2:   max = nb1
3: else
4:   max = nb2
5:
```

*pseudocode*

Traçons-le 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	

En langage Java, le *si-sinon* s'écrit :

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

Le test précédent peut donc s'écrire en supposant les variables déclarées :

```
if (nb1 > nb2){
    max = nb1;
3 } else {
    max = nb2;
}
6
```

java

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

```
algorithm exercice(a, b : integer) → integer
    integer c
    if a > b then
        c = a DIV b
    else
        c = b MOD a
    return c
```

pseudocode

```

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

```

java

▷ exerciceB(2, 3) = \_\_\_\_

▷ exerciceB(4, 1) = \_\_\_\_

```

algorithm exercice(x1, x2 : integer) → integer
    boolean ok
    ok = x1 > x2
    if ok then
        | ok = ok ET x1 == 4
    else
        | ok = ok OU x2 == 3

    if ok then
        | x1 = x1 * 1000

    return x1 + x2

```

pseudocode

```

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

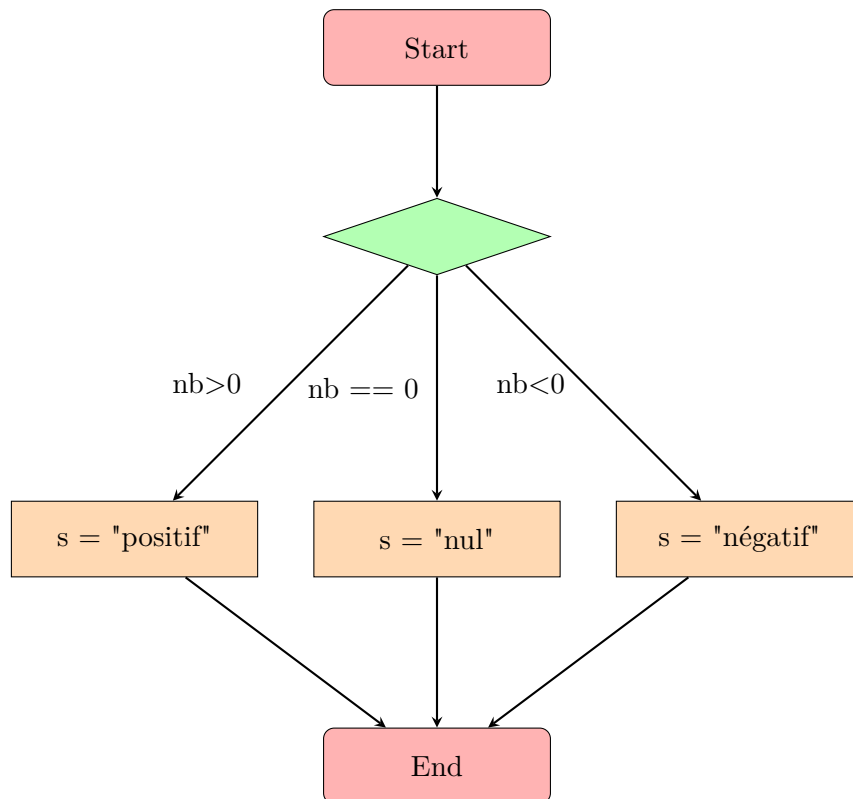
▷ exercice(2, 3) = \_\_\_\_

▷ exercice(4, 1) = \_\_\_\_

### 7.3 Le si-sinon-si

Avec cette construction, il est possible d'indiquer à un endroit de l'algorithme plus de deux chemins possibles. Partons à nouveau d'un exemple pour illustrer cette instruction.

**Exemple.** On voudrait mettre dans la chaîne `signe` la valeur "positif", "négatif" ou "nul" selon qu'un nombre donné est positif, négatif ou nul.



```

1: if nb>0 then
2:   | signe = "positif"
3: else if nb==0 then
4:   | signe = "nul"
5: else
6:   | signe = "négatif"
7:
  
```

*pseudocode*

Traçons-le

#	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"	

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

### 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.

```

if nb>0 then
|  signe = "positif"
else
|  if nb=0 then
|  |  signe = "nul"
|  else
|  |  signe = "négatif"

```

pseudocode

- ▷ 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.



```

if quantité<10 then
    prixUnitaire = 10
else if quantité<100 then           // On sait que ce n'est pas <10
                                     // inutile de le tester
    prixUnitaire = 8
else
    prixUnitaire = 6

```

*pseudocode*

## 7.4 Expression booléenne

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

**Dénition.** 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 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)

|| &&

## 7.5 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).

**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.

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

*pseudocode*

remplace avantageusement

```
if dayNumber=1 then
  dayName = "lundi"
else if dayNumber=2 then
  dayName = "mardi"
else if dayNumber=3 then
  dayName = "mercredi"
else if dayNumber=4 then
  dayName = "jeudi"
else if dayNumber=5 then
  dayName = "vendredi"
else if dayNumber=6 then
  dayName = "samedi"
else
  dayName = "dimanche"
```



*pseudocode*

En langage Java, le *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 ;
- ▷ *Statement* peut être une instruction ou plusieurs ;
- ▷ *SwitchLabels* (avec un *s*) se sont plusieurs « **case** »

Le *switch* précédent s'écrit :

```
1 switch (dayNumber) {  
    case 1:  
        dayName = "lundi";  
4 case 2:  
        dayName = "mardi";  
    case 3:  
7        dayName = "mercredi";  
    case 4:  
        dayName = "jeudi";  
10 case 5:  
        dayName = "vendredi";  
    case 6:  
13        dayName = "samedi";  
    case 7:  
        dayName = "dimanche";  
16 }
```

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.

La syntaxe générale est :

```
switch expression
| liste1 de valeurs séparées par des virgules :
|   Instructions
| liste2 de valeurs séparées par des virgules :
|   Instructions
| ...
| listek de valeurs séparées par des virgules :
|   Instructions
default :
|   Instructions
```

*pseudocode*



# Chapitre 8

## Module et références

### Contenu

8.1	Décomposer le problème . . . . .	<b>79</b>
8.2	Exemple . . . . .	<b>80</b>
8.3	Paramètres et valeur de retour . . . . .	<b>81</b>
8.3.1	Le paramètre en entrée . . . . .	82
8.3.2	Le paramètre en entrée-sortie . . . . .	83
8.3.3	La valeur de retour . . . . .	84
8.4	Type primitif et type référence . . . . .	<b>84</b>
8.4.1	Type primitif . . . . .	85
8.4.2	Type référence . . . . .	87
8.4.3	Les paramètres en Java . . . . .	87
8.5	Résumons . . . . .	<b>88</b>

### 8.1 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

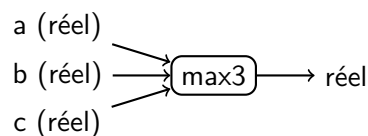
y en a peu. Par exemple, `dizaine(nb)` est plus parlant que `nb MOD 100 DIV 10` pour calculer les dizaines d'un nombre.

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

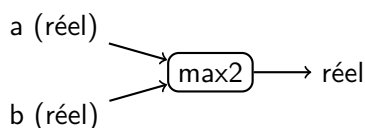
Un algorithme qui résout une partie de problème est parfois appelé **fonction**, **procédure**, **méthode** ou encore **module** en fonction du langage et du contexte. Il y a quelques nuances mais elles importent peu ici.

## 8.2 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.



```

algorithm max2(a : real, b : real)
  → real
  real max
  if a > b then
    max = a
  else
    max = b
  return max
  
```

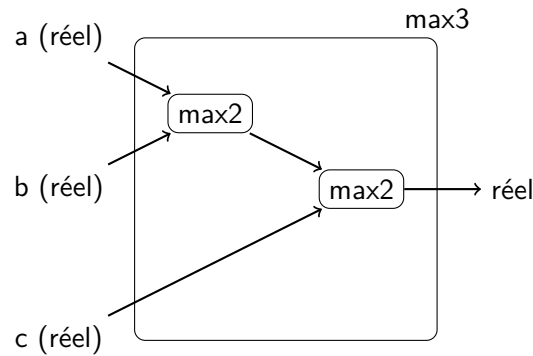
*pseudocode*

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>44</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 :

```

algorithm max3(a : real, b : real, c : real) → real
    reals maxab, max
    maxab = max2(a,b)
    max = max2(maxab,c)
    return max

```

*pseudocode*

qui peut se simplifier en :

```

algorithm max3(a,b,c : reals) → real
    return max2( max2(a,b) ,c)

```

*pseudocode*

## 8.3 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.

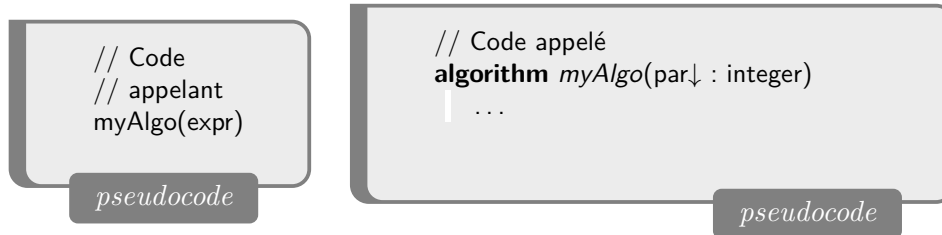
Il s'agit d'une situation fréquente mais pas obligatoire que nous pouvons généraliser. En algorithmique nous allons traiter avec trois sortes de paramètres. Nous verrons ensuite que tous les langages n'acceptent pas tous les types de paramètres.

44. 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.

### 8.3.1 Le paramètre en entrée

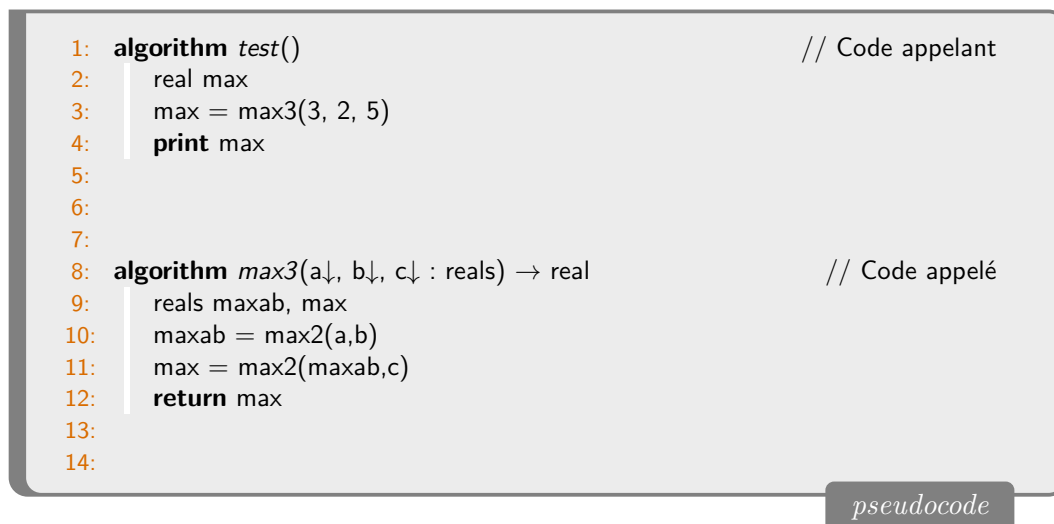
Le paramètre en **entrée** est ce que nous connaissons déjà. Il correspond à une donnée de l'algorithme. Une valeur va lui être attribuée en début d'algorithme et elle ne sera pas modifiée. On pourra faire suivre le nom du paramètre d'une flèche vers le bas ( $\downarrow$ ) pour rappeler son rôle mais 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`.

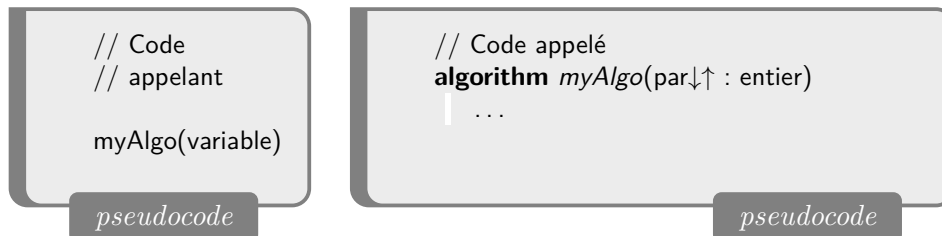
### 8.3.2 Le paramètre en entrée-sortie

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

- ▷ l'algorithme reçoit une valeur en entrée ;
- ▷ le paramètre peut être modifié.

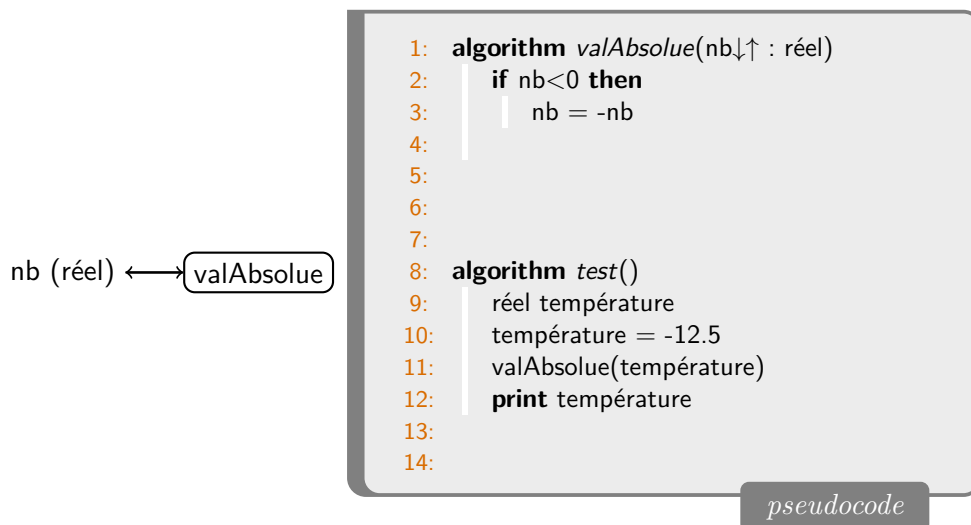
Cela signifie que l'algorithme a pour but de modifier le paramètre. Un tel paramètre sera suivi d'une double flèche ( $\downarrow\uparrow$ ).

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 (`par = variable`) et une dernière ligne pour effectuer l'assignation opposée (`variable = 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	-12.5	vrai
9	-12.5		
10, 1			
2			
3			
5, 10	12.5	12.5	

### 8.3.3 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.

## 8.4 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 nous au langage Java en commençant par faire un petit détour sur les notions de **type primitif** et **type référence**.

### 8.4.1 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>45</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.

`i`  
7

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.

Les entiers sont codés en notation en complément à deux, excepté le type `char` qui est un entier non-signé de 16 bits représentant le code Unicode codé en UTF-16 du caractère<sup>46</sup>.

Voici les tailles et les intervalles.

45. 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*).

46. Pour en savoir plus sur l'Unicode, UTF8, UTF16 et UTF32, lire « Unicode, UTF8, UTF16, UTF32... et tutti quanti »

<http://namok.be/blog/?post/2009/11/30/unicode-UTF8-UTF16-UTF32-et-tutti-quanti>

type	taille ( <i>byte</i> )	taille ( <i>bit</i> )	intervalle
byte	1	8	$[-128, 127]$ $[-2^8, 2^8 - 1]$
short	2	16	$[-32\,768, 32\,767]$ $[-2^{16}, 2^{16} - 1]$
int	4	32	$[2\,147\,483\,648, 2\,147\,483\,647]$ $[-2^{32}, 2^{32} - 1]$
long	8	64	$[9\,223\,372\,036\,854\,775\,808, 9\,223\,372\,036\,854\,775\,807]$ $[-2^{64}, 2^{64} - 1]$
char	2	16	$[0, 65\,535]$ $[0, 2^{16} - 1]$

Les nombres pseudo-réel, ou encore les nombres à virgule flottante, sont codés suivant la norme IEEE 754<sup>47</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.

$$\text{nombre} = \text{signe} \text{ mantisse}_2 2^{\text{exposant}_2}$$

type	taille ( <i>bit</i> )	exposant	mantisse
float	32	8	23
double	64	11	52

Pour les booléens, bien qu'un bit suffirait, la taille dépend de l'architecture et de la *jvm*.

47. [https://fr.wikipedia.org/wiki/IEEE\\_754](https://fr.wikipedia.org/wiki/IEEE_754)

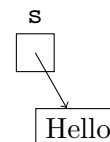
### 8.4.2 Type référence

**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.

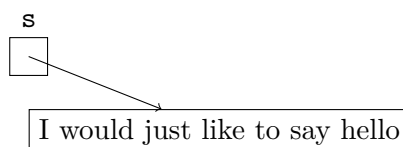
Une variable `s` de type référence et contenant la valeur "Hello" peut se représenter comme ci-contre.



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

```
1 String s = "Hello";  
  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. En effet, les types primitifs en Java sont au nombre de 8 comme nous l'avons vu tandis qu'il existe des types références prédéfinis (comme `String`) et tous les types références définis par le développeur.

Les tableaux sont aussi des types références comme nous le verrons plus tard.

### 8.4.3 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 8.3.1 (p.82) 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.

## 8.5 Résumons

Reprenons tout ce que nous venons de voir avec un exemple d'algorithme qui possède tous les types de paramètres.



```
1: algorithm testDivision()
2:   integers nb1, nb2, quotient, remainder
3:   nb1 = 5
4:   nb2 = 3
5:   quotient = division(nb1, nb2, remainder)
6:   print quotient
7:   print remainder
8:   nb1 = 7
9:   nb2 = 9
10:  quotient = division(nb1, nb2, remainder)
11:  print quotient
12:  print remainder
13:
14:
15: algorithm division( dividend↓ : integer, divisor↓ : integer,
16:   remainder↑ : integer) → integer
17:   dividend = nb1
18:   divisor = nb2
19:   remainder = remainder
20:
21:   // Le code proprement dit de l'algorithme
22:   integer quotient
23:   quotient = dividend DIV divisor
24:   remainder = dividend MOD divisor
25:
26:   remainder = remainder
27:   return quotient
28:
29:
```

*pseudocode*

Traçons-le avec : *quotient* **q**, *dividend* **D**, *divisor* **d**, *remainder* **r**.

	testDivision				division			
#	nb1	nb2	q	r	D	d	r	q
3	5							
4		3						
5, 15					5	3		
23								1
24							2	
27, 5			1	2				
8	7							
9		9						
10, 15					7	9		
23								0
24							7	
27, 10			0	7				

Pour mieux se comprendre, il est utile d'introduire un peu de vocabulaire.



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**.

Les instructions en gris dans l'exemple ne sont pas écrites mais c'est comme si elles étaient présentes pour initialiser les paramètres formels  $\downarrow$  et  $\downarrow\uparrow$  en début d'algorithme et pour donner des valeurs aux paramètres effectifs  $\downarrow\uparrow$  en fin d'algorithme.

À la fin de l'algorithme, c'est comme si la valeur retournée *remplaçait* l'appel. Dans notre exemple, c'est donc cette valeur retournée qui sera affichée.

# Chapitre 9

## Un travail répétitif

Les ordinateurs révèlent tout leur potentiel dans leur capacité à répéter inlassablement les mêmes tâches. Vous avez pu appréhender les boucles lors de votre initiation sur le site `code.org`. Voyons comment utiliser à bon escient des boucles dans nos codes.

**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.



### 9.1 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 étudiant ».

▷ 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.

## 9.2 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.**

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

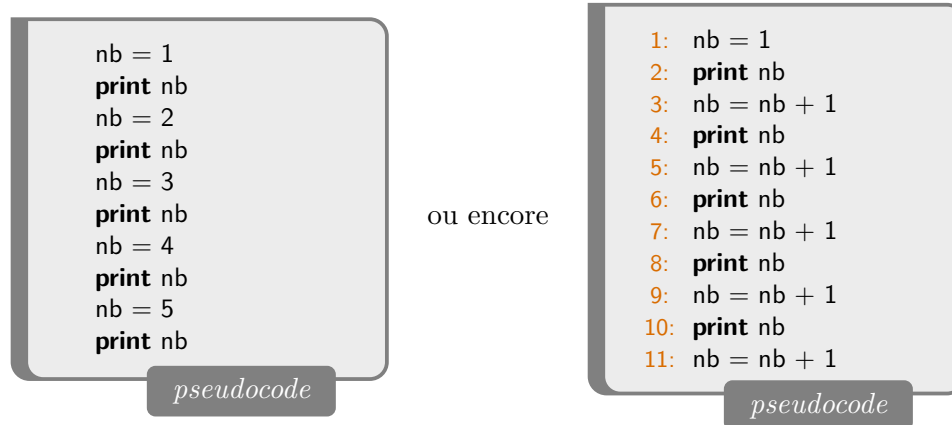
Si nous voulions un algorithme qui affiche les nombres de 1 à 5 sans utiliser de boucle, nous pourrions écrire :

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

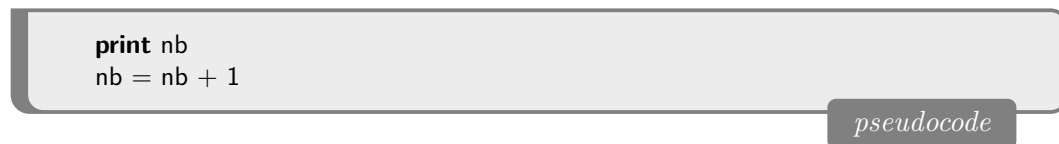
*pseudocode*

Ces cinq instructions sont proches mais pas tout-à-fait identiques. En l'état, nous ne pouvons pas encore en faire une boucle<sup>48</sup> ; il va falloir ruser. Nous pouvons obtenir le même résultat avec l'algorithme suivant :

<sup>48</sup>. 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 ?



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 :



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>49</sup>.

---

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

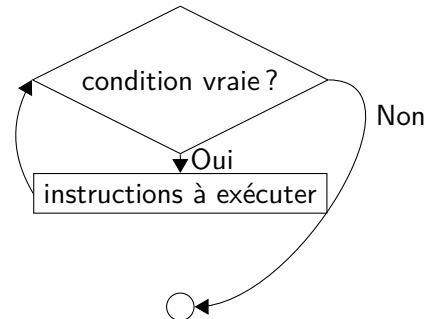
### 9.3 « tant que »

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** condition **faire**

séquence d'instructions à exécuter

**tant que**



Comme pour la structure si, la condition est une expression à valeur booléenne. Dans ce type de structure, il faut qu'il y ait dans la séquence d'instructions comprise entre tant que et fin tant que 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**. L'ordinogramme ci-dessus décrit le déroulement de cette structure. On remarquera que si la condition est fausse dès le début, la tâche n'est jamais exécutée.

#### 9.3.1 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
```

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]

```

algorithm compteur5()
  entier nb
  nb = 1
  tant que nb ≤ 5 faire
    print nb
    nb = nb + 1
  tant que
  print "nb vaut ", nb

```

#	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

### 9.3.2 Exemple - Généralisation à n nombres

On peut généraliser l'exemple précédent en affichant tous les nombres de 1 à n où n est une donnée de l'algorithme.

**algorithm** *compteur*(n : entier)

```

  entier nb
  nb = 1
  tant que nb ≤ n faire
    print nb
    nb = nb + 1
  tant que

```

### 9.3.3 Exercices

#### 1 Compréhension d'algorithmes

Quels sont les affichages réalisés lors de l'exécution des algorithmes suivants ?

<b>algorithm</b> <i>boucle1()</i> entier x x = 0 <b>tant que</b> x < 12 <b>faire</b> x = x + 2 <b>tant que</b> <b>print</b> x	<b>algorithm</b> <i>boucle2()</i> booléen ok entier x ok = vrai x = 5 <b>tant que</b> ok <b>faire</b> x = x + 7 ok = x MOD 11 $\neq$ <b>faire</b> 0 <b>tant que</b> <b>print</b> x	<b>algorithm</b> <i>boucle3()</i> booléen ok entiers cpt, x x = 10 cpt = 0 ok = vrai <b>tant que</b> ok ET cpt < 3 <b>if</b> x MOD 2 = 0 <b>then</b> x = x + 1 ok = x < 20 <b>else</b> x = x + 3 cpt = cpt + 1  <b>tant que</b> <b>print</b> x
--	--	---

#### 2 Afficher des nombres



En utilisant un **tant que**, 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 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.



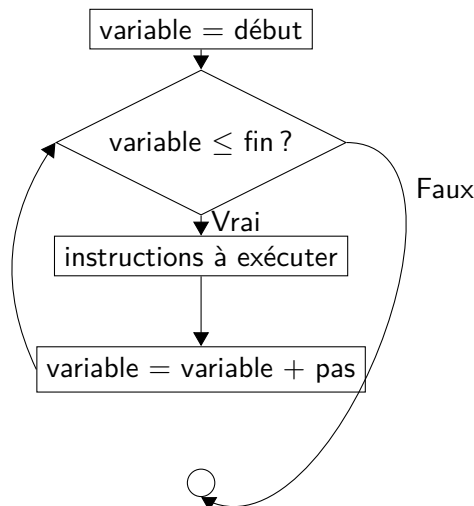
## 9.4 « pour »

Ici, on 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** variable **de** début **à** fin **par** pas **faire**

séquence d'instructions à exécuter

**pour**



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

Le pas est facultatif, et généralement omis (dans ce cas, sa valeur par défaut est 1).

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

La variable de contrôle ne servant que pour la boucle et étant forcément entière, on va considérer qu'il n'est pas nécessaire de la déclarer et qu'elle n'est pas utilisable en dehors de la boucle<sup>50</sup>.

### 9.4.1 Exemples

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



<sup>50</sup>. De nombreux langages ne le permettent d'ailleurs pas ou ont un comportement indéterminé si on le fait.

[1]

```

algorithm compterJusque5()
  // par défaut le pas est de 1
  pour nb de 1 à 5 faire
    print nb
  pour
  print "nb n'existe plus"

```

#	nb	condition	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

Si on veut généraliser l’affichage à n nombres, on a :

```

algorithm compterJusqueN(n : entier)
  pour nb de 1 à n faire
    print nb
  pour

```

### 9.4.2 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 (cf. le test dans l'organigramme ci-contre).

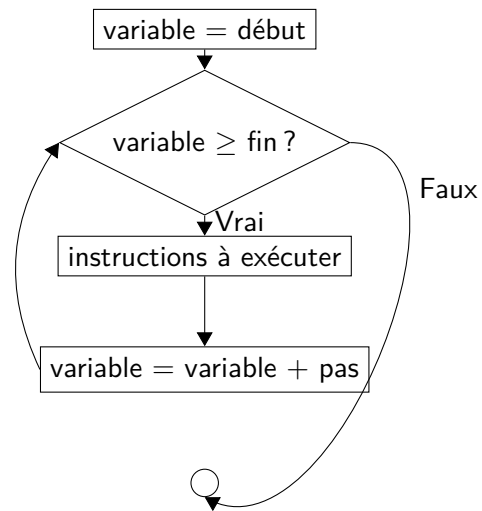
**Exemple :** Compte à rebours à partir de  $n$ .

**algorithm** *compterÀReboursDécroissant*( $n$  : entier)

```

    pour nb de  $n$  à 1 par -1 faire
        print nb
    pour
        print "Partez!"

```



### 9.4.3 Cohérence

Il faut veiller à la cohérence de l'écriture de cette structure. On considérera 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.

**Exemples :**

<b>pour</b> $i$ de 2 à 0 <b>faire</b>	// La boucle n'est pas exécutée.
<b>pour</b> $i$ de 1 à 10 <b>par</b> -1 <b>faire</b>	// La boucle n'est pas exécutée.
<b>pour</b> $i$ de 1 à 1 <b>par</b> 5 <b>faire</b>	// La boucle est exécutée 1 fois.

### 9.4.4 Modification des variables de contrôle

Il est important de ne pas modifier dans la séquence d'instructions une des variables de contrôle début, fin ou pas ! Il est aussi fortement déconseillé de modifier « manuellement » la variable de contrôle au sein de la boucle pour. Il ne faut pas l'initialiser en début de boucle, et ne pas s'occuper de sa modification, l'instruction  $i = i + \text{pas}$  étant automatique et implicite à chaque étape de la boucle.



### 9.4.5 Exemple – Afficher uniquement les nombres pairs

Cette fois-ci on affiche 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. On peut utiliser un « pour ». Une solution possible est :



**algorithm** *afficherPair*( $n$  : entier)

```

    pour nb de 2 à  $n$  par 2 faire
        print nb
    pour

```

La section sur les suites proposera d’autres solutions pour ce problème.

#### 9.4.6 Exercices

##### 3 Compréhension d’algorithmes

Quels sont les affichages réalisés lors de l’exécution des algorithmes suivants ?

**algorithm** *boucle5*()

```

entier x
booléen ok
x = 3
ok = vrai
pour i de 1 à 5 faire
    x = x + i
    ok = ok ET ( $x \text{ MOD } 2 = 0$ )
pour
if ok then
    print x
else
    print 2 * x

```

**algorithm** *boucle6*()

```

entiers fin
pour i de 1 à 3 faire
    fin = 6 * i - 11
    pour j de 1 à fin par 3 faire
        print 10 * i + j
    pour
pour

```

##### 4 Afficher des nombres



Reprenons un exercice déjà donné avec le **tant que**. En utilisant un **pour**, écrire un algorithme qui reçoit un entier  $n$  positif et affiche

- les nombres de 1 à  $n$  ;
- les nombres de 1 à  $n$  en ordre décroissant ;
- les nombres de  $-n$  à  $n$  ;
- les multiples de 5 de 1 à  $n$  ;
- les multiples de  $n$  de 1 à 100.

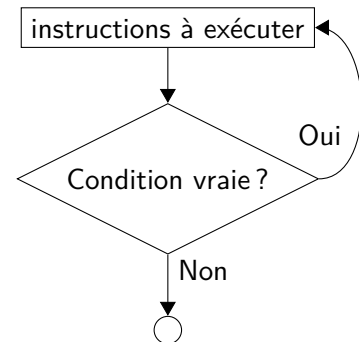
## 9.5 « faire – tant que »

Cette structure est très proche du «faire - tant que » à ceci près que le test est fait à la fin et pas au début<sup>51</sup>. La tâche est donc toujours exécutée au moins une fois.

**faire**

séquence d'instructions à exécuter

**tant que** condition



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.

### 9.5.1 Exemple

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



51. Certains langages introduisent aussi (ou à la place) un **faire jusqu'à ce que**. Dans ce cas, la boucle continue lorsque le test est **faux** et s'arrête lorsqu'il est vrai.

[1]

```

algorithm compteur5()
  entier nb
  nb = 1
  faire
    print nb
    nb = nb + 1
  tant que nb ≤ 5
  print "nb vaut ", nb

```

#	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

### 9.5.2 Exercices

#### 5 Compréhension d'algorithmes

Quels sont les affichages réalisés lors de l'exécution des algorithmes suivants ?

```

algorithm boucle4()
  booléens pair, grand
  entiers p, x
  x = 1
  p = 1
  faire
    p = 2 * p
    x = x + p
  pair = x MOD 2 = 0

```

```
grand = x > 15  
tant que NON pair ET NON grand  
print x
```

### 6 Afficher des nombres

Reprenons un exercice déjà fait avec le **tant que** et le **pour** en utilisant cette fois un **faire tant que**. 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.

## 9.6 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, on veut dire que le nombre d'itérations est déterminé au moment où on arrive à la boucle). La boucle **faire** 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. Le schéma ci-dessous propose un récapitulatif.

## 9.7 Acquisition de données multiples

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, on va devoir stocker chaque valeur donnée par l'utilisateur dans une seule et même variable et la traiter avant de passer à la suivante. Prenons un exemple concret pour mieux comprendre.

On veut pouvoir calculer (et retourner) 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.

### 9.7.1 Variante 1 : nombre de valeurs connu



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

// Lit des valeurs entières et retourne la somme des valeurs lues.

```
algorithm sommeNombres() → entier // Variante 1
    entier nbValeurs // nb de valeurs à additionner
    entier valeur // un des termes de l'addition
    entier somme // la somme
    somme = 0 // la somme se construit petit à petit. 0 au départ
    read nbValeurs
    pour i de 1 à nbValeurs faire
        read valeur
        somme = somme + valeur
    pour
    return somme
```

**7**

### 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.

### 9.7.2 Variante 2 : stop ou encore



Après chaque nombre, on demande à l'utilisateur s'il y a encore un nombre à additionner.

Ici, il faut chercher une solution différente car on ne connaît pas au départ le nombre de valeurs à additionner et donc le nombre d'exécution de la boucle. On va devoir passer à un « tant que » ou un « faire - tant que ». On peut envisager de demander en fin de boucle s'il reste encore un nombre à additionner. Ce qui donne :

// Lit des valeurs entières et retourne la somme des valeurs lues.

```
algorithm sommeNombres() → entier // Variante 2a
    booléen encore // est-ce qu'il reste encore une valeur à additionner ?
    entier valeur // un des termes de l'addition
    entier somme // la somme
    somme = 0
```



```

faire
    read valeur
    somme = somme + valeur
    read encore
tant que encore
return somme

```

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

// Lit des valeurs entières et retourne la somme des valeurs lues.

```

algorithm sommeNombres() → entier // Variante 2b
    booléen encore // est-ce qu'il reste encore une valeur à additionner ?
    entier valeur // un des termes de l'addition
    entier somme // la somme
    somme = 0
    read encore
    tant que encore faire
        read valeur
        somme = somme + valeur
        read encore
    tant que
    return somme

```

### 8 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.

#### 9.7.3 Variante 3 : valeur sentinelle

L'utilisateur entre une valeur spéciale pour indiquer la fin. On parle de valeur **sentinelle**. Ceci n'est possible que si cette valeur **sentinelle** ne peut pas être un terme



valide de l'addition. Par 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.

```
// Lit des valeurs entières positives et retourne la somme des valeurs lues.

// La valeur sentinelle est -1.

algorithm sommeNombresPositifs() → entier // Variante 3

    entier valeur // un des termes de l'addition

    entier somme // la somme

    somme = 0

    read valeur

    tant que valeur ≠ -1 faire

        somme = somme + valeur

        read valeur // remarquer l'endroit où on lit une valeur.

    tant que

    return somme
```

## 9 Choix de la valeur sentinelle

Quelle valeur sentinelle prendrait-on pour additionner une série de cotes d'interrogations ? Une série de températures ?

## 10 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.

## 11 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.

## 9.8 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 **pour** 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 vont avec. Elles pourront se généraliser à beaucoup d'autres exemples.

### Exemple 1 - Afficher les carrés

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

Si on vous demande : "Quel est le 7<sup>e</sup> nombre à afficher?". Vous répondrez : "Facile ! C'est 7<sup>2</sup>, soit 49". Plus généralement, le nombre à afficher lors du  $i^{\text{e}}$  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 :

**algorithm** *suiteCarrés*( $n$  : entier)

**pour**  $i$  de 1 à  $n$  **faire**

**print**  $i^2$

**pour**

Dans cette solution, la variable de contrôle compte simplement le nombre d'itérations. On calcule le nombre à afficher en fonction 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  $i$ . On l'appelle aussi « 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  $i^{\text{e}}$  place, souvent appelé le  $i^{\text{e}}$  terme, peut être déterminé directement au départ d'une fonction de l'indice  $i$ .

### 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...

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

Comme on peut 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.

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

Sauf pour le premier, qui ne peut pas être calculé en fonction du précédent. Une solution élégante et facilement adaptable à d'autres situations est :

**algorithm** *suite*(n : entier)

entier val

val = *1<sup>re</sup> valeur à afficher*

**pour** i de 1 à n **faire**

**print** val

val = *la valeur suivante calculée à partir de la valeur courante et/ou de précédentes valeurs*

**pour**

qui, dans notre exemple précis, devient :



// affiche la suite 1 2 4 7 11 16 ...

**algorithm** *suitePasCroissants*(n : entier)

entier val

val = 1

**pour** i de 1 à n **faire**

**print** val

val = val + i

**pour**

Cet exemple illustre un second modèle de génération de suite dans lequel l'élément à la  $i^{\text{e}}$  place, le  $i^{\text{e}}$  terme, est exprimé en fonction du ou des précédents termes. On parle dans ce cas d'une « suite récurrente ».

**12 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 précédents, 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...

**9.9 Exercices récapitulatifs**

Pour tous ces exercices, nous vous donnons peu d'indications sur la solution à mettre en œuvre. À vous de jouer...

**13 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.



**algorithm** *lireNb*( $n$  : entier)  $\rightarrow$  entier

**Solution.** Si la valeur donnée par l'utilisateur était toujours correcte, il suffirait d'écrire :

**algorithm** *lireNb*( $n$  : entier)  $\rightarrow$  entier

```
entier val
read val
return val
```

Pour tenir compte des entrées invalides, il faut ajouter une boucle qui prévient que la valeur est mauvaise et la redemande. Et ceci, tant que c'est incorrect. Ce qui donne :

**algorithm** *lireNb*( $n$  : entier)  $\rightarrow$  entier

```
entier val
read val
tant que val < 1 OU val > n faire
    print "valeur incorrecte"
    read val
```

**tant que**

**return** val

La solution qu'on obtient est proche d'une lecture répétée avec valeur sentinelle. Dans ce cas, la valeur sentinelle est toute valeur correcte.

#### 14 Lancé de dés



Écrire un algorithme qui lance  $n$  fois un dé et compte le nombre de fois qu'une certaine valeur est obtenue.

**algorithm** *lancerDé*( $n$  : entier, val : entier)  $\rightarrow$  entier

#### 15 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,  $0! = 1$ .

#### 16 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.

#### 17 Table de multiplication



Écrire un algorithme qui affiche la table de multiplication des nombres de 1 à 10 (cf. l'exemple ci-contre).

Attention ! Nous sommes en algorithmique, ne vous préoccupez pas de la mise en page de ce qui est affiché. Ce sera une question importante quand vous traduirez l'algorithme dans un langage de programmation mais pas ici.

```
1 x 1 = 1
1 x 2 = 2
...
1 x 10 = 10
2 x 1 = 2
...
10 x 9 = 90
10 x 10 = 100
```

#### 18 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.

#### 19 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.

**20 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.

**21 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  $1 + 3 + 3 + 0 + 4 + 5 = 16$ .

**22 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  $28 = 1 + 2 + 4 + 7 + 14$ .

**23 Décomposition en facteurs premiers**

Écrire un algorithme qui affiche la décomposition d'un entier en facteurs premiers. Par exemple, 1001880 donnerait comme décomposition  $2^3 * 3^2 * 5 * 11^2 * 23$ .

**24 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.

**25 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.

**26 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] ».







## Troisième partie

### Les annexes



# Chapitre 10

## Exercices

### Contenu

10.1	Spécifier le problème, exercices . . . . .	<b>115</b>
10.2	Premiers algorithmes et programmes, exercices . . . . .	<b>117</b>
10.3	Tracer un algorithme . . . . .	<b>118</b>
10.4	Division entière . . . . .	<b>120</b>
10.4.1	Exercices récapitulatifs sur les difficultés de calcul .	120
10.5	Choix . . . . .	<b>122</b>
10.6	Exercices, modules et références . . . . .	<b>126</b>

### 10.1 Spécifier le problème, exercices

Pour les exercices suivants, nous vous demandons d'imiter la démarche décrite dans ce chapitre, à savoir :

- ▷ Déterminer quelles sont les données ; leur donner un nom et un type.
- ▷ Déterminer quel est le type du résultat.
- ▷ Déterminer un nom pertinent pour l'algorithme.
- ▷ Fournir un résumé graphique.
- ▷ Donner des exemples.

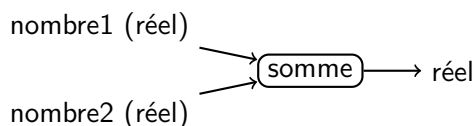
#### **1** Somme de 2 nombres

Calculer la somme de deux nombres donnés.

**Solution.**<sup>53</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

<sup>53</sup>. Nous allons de temps en temps fournir des solutions. En algorithmique, il y a souvent **plus qu'une** solution possible. Ce n'est donc pas parce que vous avez trouvé autre chose que c'est mauvais. 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.

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 :  $\text{somme}(3, 2)$  donne 5     $\text{somme}(-3, 2)$  donne  $-1$      $\text{somme}(3, 2.5)$  donne 5.5     $\text{somme}(-2.5, 2.5)$  donne 0.

## 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.

## 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 Prix TTC

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.

## 10 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.

**11 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.

**12 Somme des chiffres**

Calculer la somme des chiffres d'un nombre entier de 3 chiffres.

**13 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.

**14 Conversion secondes en heures**

Étant donné un temps écoulé depuis minuit. 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.

**15 Conversion secondes en minutes**

Étant donné un temps écoulé depuis minuit. 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.

**16 Conversion secondes en secondes**

Étant donné un temps écoulé depuis minuit. 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.

**17 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.

## 10.2 Premiers algorithmes et programmes, exercices

**18 Moyenne de 2 nombres**

Calculer la moyenne de deux nombres donnés.

**19 Surface d'un triangle**

Calculer la surface d'un triangle connaissant sa base et sa hauteur.



**20 Périimètre d'un cercle**

Calculer le périmètre d'un cercle dont on donne le rayon.

**21 Surface d'un cercle**

Calculer la surface d'un cercle dont on donne le rayon.

**22 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.

**23 Les intérêts**

Calculer les intérêts reçus après 1 an pour un montant placé en banque à du 2% d'intérêt.

**24 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.

**25 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.

**26 Prix TTC**

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.

**10.3 Tracer un algorithme****27 Tracer des bouts de code**

Suivez l'évolution des variables pour les bouts d'algorithmes donnés.

```

1: entiers a, b, c
2: a = 42
3: b = 24
4: c = a + b
5: c = c - 1
6: a = 2 * b
7: c = c + 1=0

```

*pseudocode*

#	a	b	c
1			
2			
3			
4			
5			
6			
7			

```

1: entiers a, b, c
2: a = 2
3: b = a3
4: c = b - a2
5: a =  $\sqrt{c}$ 
6: a = a / a

```

*pseudocode*

#	a	b	c
1			
2			
3			
4			
5			
6			

### 28 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: algorithm vitesseKMH(distanceM, duréeS : réels) → réel
2:   réels distanceKM, duréeH
3:   distanceKM = 1000 * distanceM
4:   duréeH = 3600 * duréeS
5:   return  $\frac{\text{distanceKM}}{\text{duréeH}}$ 
6:
7:

```

*pseudocode*

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.

#	
1	
2	
3	
4	
5	

Si vous trouvez qu'il n'est pas correct, voyez ce qu'il faudrait changer pour le corriger.

### 29 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 12,996389892 km/h.



**30 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.

**10.4 Division entière****31 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 \_\_\_\_
- ▷ 3 DIV 11 = \_\_\_\_ reste \_\_\_\_
- ▷ 11 DIV \_\_\_\_ = 2 reste 3
- ▷ \_\_\_\_ DIV 3 = 3 reste 1

**32 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 ?

```
algorithm versPrixRond(prixCentimes : entier) → entier
    return 100 - (prixCentimes MOD 100)
```

*pseudocode*

test n°	prixCentimes	réponse correcte	valeur retournée	Correct ?
1	130	70		
2	40	60		
3	99	1		
4	100	0		

**10.4.1 Exercices récapitulatifs sur les difficultés de calcul**

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.

**33 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 (entier)  $\longrightarrow$  multiple5  $\longrightarrow$  booléen

**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 :

```
1: algorithm multiple5(nombre : entier)  $\rightarrow$  booléen
2:   return nombre MOD 5 = 0
3:
4:
```

*pseudocode*

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	✓

### 34 Nombre entier positif se terminant par un 0

Calculer si un nombre donné se termine par un 0.

### 35 Les centaines

Calculer la partie *centaine* d'un nombre entier positif quelconque.

### 36 Somme des chiffres

Calculer la somme des chiffres d'un nombre entier positif inférieur à 1000.

### 37 Conversion secondes en heures

Étant donné un temps écoulé depuis minuit. 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 !

**38 Conversion secondes en minutes**

Étant donné un temps écoulé depuis minuit. 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.

**39 Conversion secondes en secondes**

Étant donné un temps écoulé depuis minuit. 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.

**40 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).

**41 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.

**42 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".

## 10.5 Choix

**43 Compréhension**

Tracez cet algorithme avec les valeurs fournies et donnez la valeur de retour.

```
algorithm exerciceA(a, b : entiers) → entier
    entier c
    c = 2 * a
    if c > b then
        c = c - b
    return c
```

pseudocode

▷ `exerciceA(2, 5) = ____`▷ `exerciceA(4, 1) = ____`**44 Simplification d'algorithmes**

Voici quelques extraits d'algorithmes corrects du point de vue de la syntaxe mais contenant des lourdeurs d'écriture. Simplifiez-les.

```
if ok = vrai then  
  print nombre
```

*pseudocode*

```
if ok = faux then  
  print nombre
```

*pseudocode*

```
if ok = vrai OU ok = faux then  
  print nombre
```

*pseudocode*

```
if ok = vrai ET ok = faux then  
  print nombre
```

*pseudocode***45 Compréhension**

Tracez ces algorithmes avec les valeurs fournies et donnez la valeur de retour.

```
algorithm exerciceB(b, a : entiers) → entier  
  entier c  
  if a > b then  
    c = a DIV b  
  else  
    c = b MOD a  
  return c
```

*pseudocode*▷ `exerciceB(2, 3) = ____`▷ `exerciceB(4, 1) = ____`

```

algorithm exerciceC(x1, x2 : entiers) → entier
    booléen ok
    ok = x1 > x2
    if ok then
        | ok = ok ET x1 = 4
    else
        | ok = ok OU x2 = 3

    if ok then
        | x1 = x1 * 1000

    return x1 + x2

```

*pseudocode*

▷ exerciceC(2, 3) = \_\_\_\_

▷ exerciceC(4, 1) = \_\_\_\_

#### 46 Simplification d'algorithmes

Voici quelques extraits d'algorithmes 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.

```

if condition then
    | ok = vrai
else
    | ok = faux

```

*pseudocode*

```

if a > b then
    | ok = faux
else
    | if a ≤ b then
        | ok = vrai

```

*pseudocode*

#### 47 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 ??.

#### 48 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 ?

**49 Fonction de Syracuse**

Écrire un algorithme qui, étant donné un entier  $n$  quelconque, retourne le résultat de la fonction  $f(n) = \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n + 1 & \text{si } n \text{ est impair} \end{cases}$

**50 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.

**51 Maximum de 3 nombres**

Écrire un algorithme qui, étant donné trois nombres quelconques, recherche et retourne le plus grand des trois.

**52 Le signe**

Écrire un algorithme qui **affiche** un message indiquant si un entier est strictement négatif, nul ou strictement positif.

**53 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.

**54 Dés identiques**

Écrire un algorithme qui lance trois dés et indique si on a obtenu 3 dés de valeur identique, 2 ou aucun.

**55 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.

**56 Numéro du jour**

Écrire un algorithme qui retourne le numéro du jour de la semaine reçu en paramètre (1 pour "lundi", 2 pour "mardi"...).



**57 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.

**58 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.

**59 Réussir DEV1**

Pour réussir l'UE (unité d'enseignement) DEV1, il faut que la cote attribuée à cette UE soit supérieure ou égale à 50%. Cette cote tient compte de votre examen intégré et de vos interrogations. Écrire un algorithme qui reçoit la cote finale (sur 100) d'un étudiant pour l'UE DEV1 et qui indique si l'étudiant a réussi cette UE.

**60 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 ?

**61 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.

**62 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.

## 10.6 Exercices, modules et références

todo : revoir les exercices avec des paramètres en sortie.

**63** Tracer des algorithmes

Indiquer quels nombres sont successivement affichés lors de l'exécution des algorithmes ex1, ex2, ex3 et ex4.

**algorithm** ()  $\rightarrow$  ex1

entiers x, y

addition(3, 4, x)

**print** x

x = 3

y = 5

addition(x, y, y)

**print** y

**algorithm** *addition*(a $\downarrow$ , b $\downarrow$ , c $\uparrow$  : entiers)

entier somme

somme = a + b

c = somme

**algorithm** ()  $\rightarrow$  ex2

entiers a, b

addition(3, 4, a) // voir ci-dessus

**print** a

a = 3

b = 5

soustraction(b, a, b)

**print** b

**algorithm** *soustraction*(a $\downarrow$ , b $\downarrow$ , c $\uparrow$  : entiers)

c = a - b

**algorithm** ()  $\rightarrow$  ex3

entiers a, b, c

calcul(3, 4, c)

**print** c

a = 3

b = 4

c = 5

calcul(b, c, a)

**print** a, b, c

**algorithm** calcul( $a \downarrow$ ,  $b \downarrow$ ,  $c \uparrow$  : entiers)

a = 2 \* a

b = 3 \* b

c = a + b

**algorithm** ()  $\rightarrow$  ex4

entiers a, b, c

a = 3

b = 4

c = f(b)

**print** c

calcul2(a, b, c)

**print** a, b, c

**algorithm** calcul2( $a \downarrow$ ,  $b \downarrow$ ,  $c \uparrow$  : entiers)

a = f(a)

c = 3 \* b



```
c = a + c
```

**algorithm**  $f(a \downarrow : \text{entier}) \rightarrow \text{entier}$

```
entier b
```

```
b = 2 * a + 1
```

```
return b
```

#### 64 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 ?

**algorithm**  $() \rightarrow \text{PGCD}a \downarrow, b \downarrow : \text{entiersentier}$

```
[1] a = PGCD(24, 32)
```

```
[2] a = PGCD(a, 24)
```

```
[3] b = 3 * PGCD(a + b, 2*c) + 120
```

```
[4] PGCD(20, 30)
```

```
[5] a = PGCD(a, b, c)
```

```
[6] a = PGCD(a, b) + PGCD(a, c)
```

```
[7] a = PGCD(a, PGCD(a, b))
```

```
[8]
```

```
read PGCD(a, b)
```

```
[9]
```

```
print PGCD(a, b)
```

```
[10] PGCD(a, b) = c
```

#### 65 Maximum de 4 nombres

Écrivez un algorithme qui calcule le maximum de 4 nombres.

**66 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.

**67 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".

**68 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 ??).

**69 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.

**70 Généraliser un algorithme**

Dans l'exercice ??, 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 lasseriez bien vite.

N'est-il pas possible d'écrire un seul algorithme, plus général, qui résolve tous ces problèmes d'un coup ?

