

# **PRINCIPES ALGORITHMIQUES ET PROGRAMMATION**

**Apprendre à programmer en passant par Java et Python**

Van Oudenhove Didier

12 mars 2024



## Algorithmme

Dans le cadre de ce cours, nous allons apprendre à programmer, c'est à dire partir d'un problème, l'exprimer dans un langage de programmation pour pouvoir l'exécuter par la suite sur un ordinateur. On va donc partir d'un énoncé, le problème à programmer, on va tâcher de le comprendre et de trouver une façon de le résoudre en l'exprimant par une suite d'actions qui seront transcris dans un langage de programmation. Bien évidemment dans le cadre de ce cours, on va commencer par des exemples simples pour pouvoir apprendre au fur et à mesure les différents « outils » que l'on retrouve dans les langages de programmation traditionnels comme les variables et constantes, l'assignation, les ruptures de séquence, les boucles, les procédures et fonctions,... Nous allons regarder aussi comment stocker en mémoire l'information, comment exprimer une suite de nombres, un tableau, une liste, une arborescence, un graphe,...

Après avoir vu les principaux concepts, on va pouvoir les utiliser pour traiter des problèmes plus complexes où la notion d'algorithmique prendra tout son sens, ce sera un peu comme un jeu de stratégie où il y a plusieurs façons de procéder avec des bonnes et des moins bonnes solutions. En informatique, il va falloir penser à plusieurs aspects comme la rapidité d'exécution, l'économie de mémoire, l'économie d'énergie, d'avoir bien entendu une solution exacte ou suffisamment exacte en fonction des sujets. Bien entendu, si on vous demande de trier par ordre croissant une suite de nombres, votre algorithme devra donner dans tous les cas une solution exacte, il existe énormément d'algorithmes de tri mais même si certains sortent du lot par rapport à d'autres comme le « quick-sort », il n'y a pas un algorithme parfait pour toutes les situations, cela dépendra de la quantité de nombres à trier, des nombres eux-mêmes, du fait que les données sont déjà en partie triées ou non, en fonction qu'il existe des doublons ou non,...

Mais pour certains problèmes la solution optimale est tout simplement impossible à trouver dans un temps fini mais une solution approchée pourrait, sans être la meilleure, être une solution satisfaisante. Pour illustrer ce dernier point, je vais reprendre un exemple que j'ai du traiter voilà déjà plusieurs années.

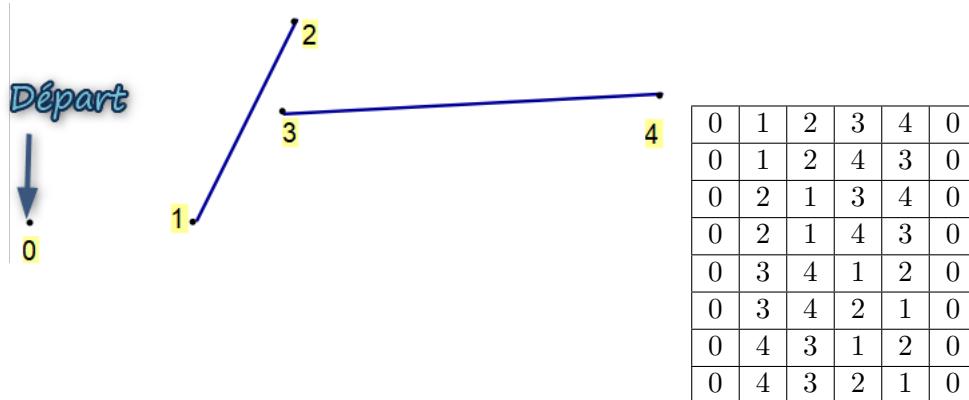
Vous avez peut-être déjà entendu parlé d'une table traçante, les architectes les utilisent pour imprimer des grands plans. Elle est constituée d'un tourniquet comprenant plusieurs stylets de couleur différente et généralement d'un axe qui fait bouger la feuille dans un sens et un autre axe pour déplacer le stylet courant, le dernier mouvement consiste à descendre le stylet sur le papier pour écrire. Vous pouvez facilement imaginer le temps nécessaire pour dessiner un plan complexe, il est donc capital d'optimiser au maximum le fichier comprenant les différentes étapes du tracé.

La première étape va consister à comprendre la problématique et ensuite d'identifier une méthode pour optimiser le temps de tracé.

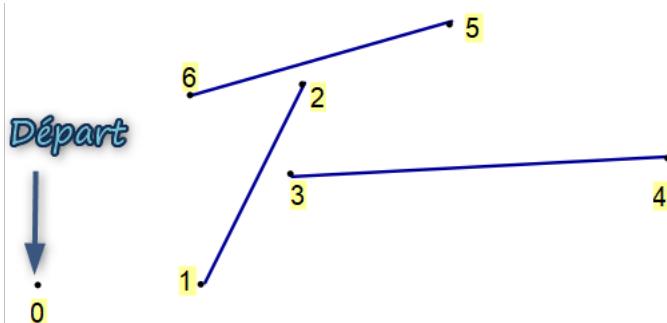
Il est tout à fait possible d'utiliser une technique bulldozer qui consiste à tester toutes les combinaisons possibles et de ne garder qu'une des meilleures solutions, cette stratégie consiste à appliquer un algorithme bien connu « le *backtracking* », même s'il n'est pas très difficile à mettre en œuvre, il coûtera très cher en temps de calcul à un tel point qu'il deviendra vite inexploitable lorsque le nombre

de segments augmentera. On parle de complexité d'un algorithme qui consiste à étudier le nombre d'opérations à effectuer en fonction de la taille du problème, ici ce serait le nombre de chemins que l'on devrait tester en fonction du nombre de segments.

Regardons les différentes possibilités avec seulement 2 segments, le stylet part du point 0 et devra, après avoir dessiné les 2 segments, revenir au même point de départ:



On peut donc compter 8 chemins, même si chaque chemin possède son inverse avec un temps de parcours similaire. Allons un pas plus loin et rajoutons un troisième segment, à votre avis combien de chemins existe-t-il? A partir du point 0, on peut opter pour 6 possibilités ensuite on se retrouve avec chaque fois le même nombre de possibilités qu'avec 2 segments càd 8, on fait vite le calcul:  $6 * 8$  qui nous donne 48 chemins possibles.



Si on exprime ce nombre de chemins en fonction du nombre de segments soit  $n$  avec  $n > 0$  on obtient :

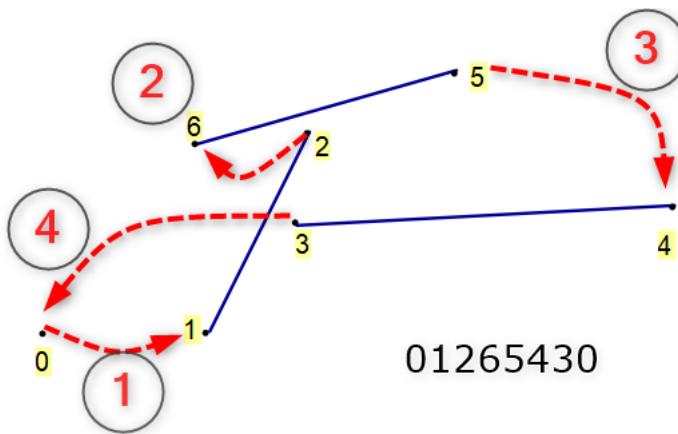
$$\text{nombre de chemins : } (n! * 2^n)$$

Imaginons un petit dessin avec une centaine de droites, ce qui donnerait :  $100! * 2^{100} \simeq 1,183 * 10^{188}$  chemins à étudier.

Bien entendu, on pourrait éliminer un chemin dès qu'il devient trop long, cependant le temps que notre algorithme sorte la bonne solution, notre dessin aura déjà eu le temps de s'imprimer de nombreuses fois sans aucune optimisation. Conclusion cet algorithme n'a aucun intérêt pour ce problème!

Il est par contre tout à fait possible de trouver une solution satisfaisante très rapidement sans qu'elle soit la solution optimale, par exemple en choisissant à chaque fois le prochain segment le plus proche non traité:

Ce qui nous donnerait sur l'exemple précédent:



Comme on peut le constater par le biais de cet exemple, après avoir compris l'objectif, il faut trouver une façon de procéder qui est réalisable et qui va apporter une solution satisfaisante.

L'étape suivante va consister à écrire l'algorithme sous forme d'une suite d'actions. Pour rester agnostique à un langage particulier et pour pouvoir exprimer son algorithme, j'aime bien commencer à exprimer celui-ci par une sorte de macro-algorithme qui va mettre en évidence les grandes étapes en utilisant des mots en français et des structures élémentaires comme des ruptures de séquence, des boucles,...



# Chapitre 2

## Pseudo-code et langage de programmation

La programmation d'un problème commence par sa compréhension. Cette compréhension permet de définir un algorithme qui pourra ensuite être implémenté dans un langage donné.

L'algorithme est indépendant du langage de programmation de ce fait, nous allons d'abord exprimer un problème sous forme d'un pseudo-code qui permet plus de liberté et que l'on traduira ensuite en Java et en Python.

Pourquoi Java et Python, actuellement la plupart des cours d'initiation à la programmation utilisent Python. Python est un langage en vogue cependant il est très différents des langages traditionnels comme C, C++, C#, Java. Étant donné que vous aurez à faire à ces 2 langages dans la suite de votre cursus, il m'a semblé intéressant de les étudier sans pour autant les approfondir.

Vous trouverez un résumé du pseudo-code dans les annexes: (18.1)



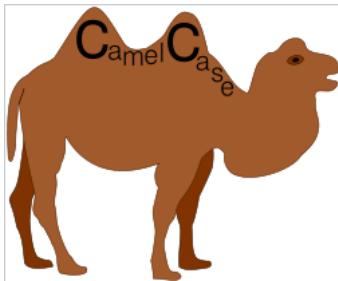
# Chapitre 3

## Notion de variable, de constante et de type

### 3.1 La notion de variable

Un programme doit pouvoir manipuler de l'information, généralement des données sont fournies en entrée, elles sont ensuite traitées par le programme qui va fournir un ou plusieurs résultats en sortie. Cette information devra être stockée dans ce qu'on appelle des *variables*. Une variable sera identifiée par un nom, plus généralement appelé un identificateur:

**L'identificateur** est un nom donné à un objet manipulé au sein d'un programme, comme ici à une variable. Généralement ce nom sera le plus explicite possible pour désigner son contenu, il commencera généralement par une lettre (pas par un chiffre en tout cas), ne contiendra pas d'espace ni d'accent et suivra souvent la notation *CamelCase*. La première lettre d'une variable sera généralement en minuscule.



Exemple: nomClient, i, nbPassages

**La visibilité** un programme est souvent organisé en blocs de code. Un bloc peut être une fonction, un objet, un module,... Il est possible d'avoir plusieurs variables de même nom au sein de blocs de code différents. Il est un peu tôt pour parler de ces aspects plus complexes qui changent aussi en fonction du langage utilisé. Dans le cadre d'une programmation procédurale, la règle est généralement qu'une variable définie au sein d'un bloc n'est pas visible en dehors de celui-ci et qu'elle sera accessible à ses blocs internes si ceux-ci ne redéfinissent pas une variable de même nom. Dans un environnement orienté objet, c'est plus complexe car une variable comme une méthode (procédure au sein d'un objet) sont encapsulées dans un objet et peuvent avoir une visibilité (privée, publique, protégée, ...).

**La durée de vie** une variable a une durée de vie qui va dépendre de l'endroit où elle est déclarée et initialisée. En règle générale, une variable déclarée au sein du programme principal sera définie lors du chargement du programme et sera détruite lors de sa clôture, c'est ce qu'on appelle une **variable globale**. Dans le cadre d'un programme compilé, ces variables sont connues au chargement du programme et un espace de taille adaptée sera réservé. Une variable déclarée au sein d'une fonction n'existera qu'au moment où la fonction est appelée et sera détruite au retour de la fonction, c'est ce qu'on appelle une **variable locale**, celle-ci sera créée sur le *stack*. Il est

également possible de demander dynamiquement de la mémoire par exemple dans un langage procédural comme le langage C on retrouve l'instruction « *malloc* », en Pascal l'instruction « *GetMem* » mais aussi dans tous les langages orientés objet, on crée dynamiquement de la mémoire lors de l'instanciation d'un objet! La réservation de **mémoire dynamique** se fait dans une zone appelée le **TAS** ou le *heap* en anglais.

**La stabilité** une variable sous entend qu'elle est modifiable. Quand une variable n'est pas modifiable, on parlera de constante. C'est techniquement un peu plus subtile car on retrouve la vraie constante qui n'est en réalité pas stockée en mémoire et les variables non modifiables qui elles sont stockées en mémoire.

```
const PI=3.14      /* PI n'existera plus à l'exécution */
...
circonf ← 2 * PI * rayon    /* sera codé par 6.28*rayon */
surface ← 2 * PI * rayon2  /* sera codé par 6.28*rayon au carré */
...
```

**Le type** une variable a toujours un type, dans certains langages il faut préciser le type de la variable avant de l'utiliser, c'est le cas des langages traditionnels comme le C, le Pascal, Java, C#, C++,... mais par exemple en Python, le type de la variable sera déduit en fonction de la donnée assignée. Le type a beaucoup d'importance car il va imposer les aspects suivants:

- ☛ la taille prise en mémoire
- ☛ La manière dont sera codée en mémoire la variable
- ☛ les opérations que l'on pourra effectuer dessus
- ☛ le formatage des données lors de l'assignation et l'affichage
- ☛ les contraintes associées

**L'initialisation** une variable doit être initialisée avant de pouvoir l'utiliser pour éviter des résultats indésirables. Là aussi dans certaines situations et pour certains langages de programmation, une variable aura une valeur initiale en fonction de son type, généralement 0 pour les valeurs numériques, une chaîne de caractères vide pour les chaînes de caractères (string), une adresse à zéro « *null* » pour les pointeurs ou référents. Mais la plupart du temps, elles ne seront pas initialisées et auront donc une valeur quelconque. Il est souvent plus prudent d'initialiser les variables et de toute façon bien connaître le comportement propre à chaque langage.

Initialisation des variables en Java		Initialisation par défaut
Variables locales	paramètres de fonction	initialisé lors de l'appel de la fonction
	autres variables locales	jamais initialisées
Variables d'objet et de classe	boolean	false
	byte, short, int, long	0
	float, double	0.0
	char	'\u0000'
	Référent (adresse)	null

## 3.2 Le tas et la pile

Toutes les variables locales (au sein d'une procédure ou fonction) sont créées sur la **pile** et seront supprimées de celle-ci lorsque la procédure se terminera.

Toutes les variables qui sont demandées dynamiquement, comme la création d'un objet, d'une demande de zone de mémoire, seront créés sur le « tas » et seront libérées soit par l'utilisateur soit par le « garbage collector »<sup>1</sup>.

### 3.3 La notion de constante

Comme on l'a vu plus haut (3.1), dans certains langages et plus particulièrement dans les compilateurs, une constante n'existe plus à l'exécution car elle est remplacée partout dans le code par sa valeur. Cependant ce n'est pas toujours le cas car la constante peut aussi être une variable qu'on indique comme non modifiable et pire encore en Python le concept de constante n'existe pas.

Généralement l'identificateur d'une variable « constante » se note en majuscule où chaque mot est séparé par un caractère souligné:

```
Const PI = 3.141592
Const NB_CARTES=52
```

---

#### Algorithme 3.1 Création d'une constante

---

 *python* → *l'identificateur en majuscule*  
`PI=3.141592`

 *//En Java on préfixe la variable par le mot réservé "final"*  
`final float PI=3.141592`

---

### 3.4 La notion de type

Une variable contiendra une information qui sera codée en mémoire et qui occupera un place exprimée en un nombre d'octets. En fonction du type de la variable, on pourra effectuer certaines opérations et l'affichage sera formaté en conséquence.

Voici par exemple plusieurs façons de coder le nombre 23:

- 23 codé comme un entier sur 8 bits

0	0	0	1	0	1	1	1
7	6	5	4	3	2	1	0

- 23 codé comme un entier sur 16 bits

0	0	0	0	0	0	0	0	0	0	1	0	1	1	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

- 23 codé comme un réel sur 32 bits

0	1	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	0	0	...	0	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10

- 23 codé comme une chaîne de caractères en utilisant l'encodage ASCII:

0	0	1	1	0	0	1	0	0	0	1	1	0	0	1	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Il est aussi évident que l'on ne pourra pas effectuer les mêmes opérations sur des nombres que sur une chaîne de caractères:

- Garbage Collector: Le ramasseur d'ordures est un processus qui se charge de trouver les objets inaccessibles et de les détruire. Java comme Python possède un "garbage collector".

**avec une valeur**  $23+2 ==> 25$

**avec du texte:** "23"+"2" ==> "232" (concaténation des 2 textes)

Un type est une sorte de masque que l'on applique sur une zone mémoire pour savoir comment l'interpréter et y accéder.

### 3.4.1 Les types primitifs

Les types primitifs dans les langages de programmation sont généralement les types entier, réel, booléen, caractère et adresse mémoire (pointeur). Attention, dans certains langages de programmation comme Python, il n'existe pas de type primitif car ils sont tous enveloppés dans un objet, ainsi une variable de type entier en Python est en réalité une variable qui contient une adresse vers un objet (on parle de référent car elle référence un objet). On abordera la notion de « classe » et « d'objet » un peu plus loin.

Voici les principaux types élémentaires:

Catégorie	Type	Taille	Description
Entier non signé	<i>byte</i>	8	entier allant de [0, 255]
	<i>word</i>	16	entier allant de [0, $2^{16}$ [
Entier signé	<i>short</i>	16	entier allant de $[-2^{15}, +2^{15}]$
	<i>int</i>	32	entier allant de $[-2^{31}, +2^{31}]$
Réel IEEE754	<i>long</i>	64	entier allant de $[-2^{63}, +2^{63}]$
	<i>float</i>	32	réel sur 32 bits
Caractère	<i>double</i>	64	réel sur 64 bits
	<i>char</i>	1 ou 2	un caractère codé sur 8 bits ou 16bits en fonction de l'encodage
Booléen	<i>String</i>	1 ou 2 *nb Car	un vecteur de caractères. Mais il s'agit généralement d'un vecteur de caractères ou d'une classe qui enveloppe un vecteur de caractères. Donc pas toujours un type primitif.
	<i>boolean</i>	1 ou autre	ne peut contenir généralement que vrai ( <i>true</i> ) ou faux ( <i>false</i> ), mais dans certains langages, un nombre entier à 0 est considéré comme <i>false</i> et les autres valeurs comme <i>true</i> .
Pointeur	Pointer	32 ou autre	adresse d'une zone mémoire, d'un objet

TABLEAU 3.1 – Types primitifs

En Java chaque type primitif possède une classe correspondante comme le montre la tableau suivant.

Types primitifs et classes associées	
Type primitif	Classe associée
byte	Byte
int	Integer
short	Short
long	Long
boolean	Boolean
float	Float
double	Double
char	Character

Une variable de type « *int* » consommera moins de mémoire qu'un objet de type « *Integer* », mais dans certains cas, on a besoin d'avoir des objets comme on le verra plus loin.

En Java, il existe un mécanisme qui facilite l'assignation entre un type primitif et sa classe associée et inversement, on parle d'*auto-boxing* et de *unboxing*:

```
//Exemple d'auto-boxing
Integer i=27; //à la place de: Integer i=new Integer(27)
//Exemple d'unboxing
int j=i; //à la place de: int j = i.intValue()
```

### 3.4.2 Un type énuméré

Un type énuméré est un type qui permet de définir une liste finie d'identificateurs où chaque identificateur est associé à une valeur allant généralement de  $0 \rightarrow (n - 1)$ , mais ce n'est pas implémenté de la même façon d'un langage à l'autre.

Par convention, chaque élément énuméré est noté en majuscule comme une constante.

---

#### Algorithme 3.2 Exemple de type énuméré (pseudo-code)

```
enum Jour = {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE}
jour1 = Jour.LUNDI      /* jour1 contiendra LUNDI mais codé par 0 */
jour2 = Jour.VENDREDI   /* jour2 contiendra VENDREDI mais codé par 4 */
if (jour1 < jour2) then  /* 0 est plus petit que 4 */
    print “{jour1} est avant {jour2}”    /* Affichera LUNDI est avant VENDREDI */
else print “{jour1} est après ou égal à {jour2}”
```

---

Une variable de type énumérée est efficace car derrière chaque identificateur se cache une valeur entière.

Regardons un exemple en Java et en Python.

### Algorithme 3.3 Création d'une constante

#En Python, on utilise une classe qui hérite de Enum, où chaque élément est précisé par une valeur

```
from enum import Enum
class Couleur(Enum):
    ROUGE = 0 #Ici on peut préciser d'autres valeurs que 0, 1 , 2
    VERT   = 1
    BLEU   = 2
```

//En java chaque élément énuméré sera un objet de la classe  
↪ Couleur

```
public enum Couleur{ROUGE, VERT, BLEU};
```

Les 2 langages permettent des structures plus complexes mais ces aspects sortent du cadre de ce cours.

#### 3.4.3 Un référent vers un objet

Un référent est une variable qui va contenir l'adresse d'un objet, le type du référent permet de s'assurer que l'on ne pourra assigner à cette variable qu'un objet de la classe du référent ou d'une classe qui hérite de celle-ci.



#### Une règle importante en programmation orienté objet

Qui peut le plus peut le moins!

Prenons en exemple une classe *MédecinGénéraliste* et une classe *MédecinSpécialiste* qui est une classe spécialisée de la première. Un médecin spécialiste est avant tout un médecin généraliste qui possède de nouvelles compétences voir des spécialisations des compétences d'un généraliste.



Considérons une variable du type *MédecinGénéraliste*, cette variable est donc un référent qui pointera vers un objet *MédecinGénéraliste* ou un *MédecinSpécialiste* car celui-ci est compatible puisqu'il peut réaliser les tâches d'un généraliste. Cependant tant qu'un objet n'est pas assigné à cette variable, il sera à vide « null ».

Il faudra s'assurer que l'objet est bien présent avant d'appeler ses méthodes pour éviter un plantage manifeste de l'application! On doit dans certain cas s'assurer que la variable référence bien un objet avant d'appeler une de ses méthodes:

---

**Algorithme 3.4** Référent vers un objet

---

```
//poste est un référent
MedecinGeneraliste poste; // contient une adresse null
...
//teste le référent avant de l'utiliser, du moins s'il risque d'être à null
if (poste !=null) poste.prendreTension();
```

---

Reprenons notre référent de type *MédecinGénéraliste*, ce référent pourra référencer les méthodes « *suturer* » et « *prendreTension* » mais pas la méthode « *opérer* ». Par contre, si on assigne à cette variable un médecin spécialiste, lorsqu'on appelle la méthode « *suturer* », ce sera bien la méthode du médecin spécialiste qui sera exécutée!

**3.4.4 Un référent vers une fonction**

Dans certains langages, une variable peut également contenir une fonction, on peut ainsi imaginer appeler une fonction à partir d'une variable et avoir ainsi des comportements différents d'une assignation à l'autre.

Imaginons une variable « *f* » qui accepte une fonction de la forme « *f(x)* » , on ne parle pas d'une variable qui recevra le résultat de l'appel de la fonction mais bien la fonction elle-même!

---

**Algorithme 3.5** Assignation d'une fonction en Python

---



```
f= lambda x:2*x*x-3*x+1
print(f(2.0))
#Affichera: 3.0
```

---

En Java, une variable peut également recevoir une fonction mais il faut avant tout définir son type sous forme d'une interface fonctionnelle, càd une interface avec une seule méthode. Une interface en Java doit être vue comme un contrat (ensemble de fonctions) qui devra être satisfait par la classe qui l'implémente. Il existe déjà plusieurs interfaces prédéfinies dans la librairie « *java.util.function* » dont la liste se trouve à l'adresse:

<https://docs.oracle.com/javase/8/docs/api/java/util/function-package-summary.html>

Par exemple l'interface « *DoubleFunction<R>* », possède une fonction « *apply* » qui reçoit en entrée un paramètre de type générique « *R* » et en retour un *Double*.

---

**Algorithme 3.6** Assignation d'une fonction en Java

---



```
DoubleFunction<Double> f= (x)->2*x*x-3*x+1;
System.out.println(f.apply(2.0));
// ==> affichera 3.0
```

---

**3.5 Définir une valeur immédiate**

Maintenant que vous savez ce que sont un type et une variable, regardons comment exprimer une valeur immédiate dans les langages de programmation.

D'abord, il ne faut pas oublier que toute information est codée en mémoire avec des « 1 » et des « 0 », donc sous forme binaire, cependant les valeurs que vous précisez dans un programme sont par défaut en base 10. Par moment un programmeur a besoin de définir des masques qui seront constitués de 0 et 1 à des endroits bien précis, pour ce faire, il est beaucoup plus pratique de pouvoir les exprimer en base 2 ou en base 16. Ainsi la plupart des langages de programmation permettent de définir une valeur immédiate en base 10, 2, 8 ou 16:

Base	Préfixe	Exemple
Décimale		178
Binaire	0b	0b10110010
Octale	0o	0o262
Hexadécimale	0x	0xb2

Les langages offrent généralement des méthodes pour transformer une valeur en une chaîne de caractères pour la représenter en binaire, en octal ou en hexadécimal:

	Bases	Méthodes	Résultat
	Binaire	Integer.toBinaryString(178)	10110010
	Octale	Integer.toOctalString(178)	262
	Hexadécimale	Integer.toHexString(178)	b2
	Base 2 ⇒ 36	Integer.toUnsignedString(-1,16)	fffffff

	Bases	Méthodes	Résultat
	Binaire	bin(178)	0b10110010
	Octale	oct(178)	0o262
	Hexadécimale	hex(178) hex( -1 )	0xb2 -0x1

## Les ruptures de séquence

Une programme est constitué d'une suite d'instructions qui s'exécutent séquentiellement jusqu'au moment où cette séquence est interrompue par une instruction qui va poursuivre l'exécution à un autre endroit. On peut distinguer plusieurs types de rupture de séquences:

- ☛ Le saut inconditionnel:

Lorsque le programme rencontre une telle instruction, le programme va poursuivre son exécution à une adresse (une position) désignée généralement par une marque qu'on appelle un **label**. Il est assez rare d'utiliser ce type de saut dans les langages de haut niveau

- ☛ Le saut conditionnel:

Le test le plus utilisé qui consiste à exécuter certains instructions si une condition est validée, il existe plusieurs variantes.

- ☛ La boucle conditionnelle:

L'idée est de répéter une suite d'instructions jusqu'à ce que ou tant qu'une condition est vérifiée. On peut voir ceci comme une variante de saut conditionnel mais avec un retour en arrière au lieu d'une poursuite en avançant dans le programme. Il existe aussi plusieurs variantes: la boucle « *tantque* », la boucle « *pour* » ou des variantes de celles-ci.

- ☛ Le retour d'une procédure ou d'une fonction. A la fin d'une procédure ou d'une fonction, on poursuit l'exécution du programme après l'instruction qui a appelé la procédure ou la fonction.

- ☛ Rupture suite à une exception. Lorsqu'une erreur est détectée, il est possible que celle-ci provoque le déclenchement d'une exception qui va dévier l'exécution du programme vers un bloc de code qui aura pour tâche de gérer cette erreur ou d'arrêter le programme si celle-ci n'est pas récupérable.

### 4.1 Les différents types de sauts

#### 4.1.1 Le saut inconditionnel

Les structures de sauts inconditionnels n'existent ni en Python ni en Java. La structure du saut inconditionnel est courante en langage d'assembleur avec l'instruction « `jmp unLabel` » ou dans le langage *Basic* avec la fameuse instruction « `GOTO unLabel` » très critiquée mais tout de même présente dans plusieurs langages. Cette structure est très décriée car elle amène souvent à avoir un programme illisible avec des sauts dans tous les sens, on parle de programmation *spaghetti*.

**Algorithme 4.1** Le saut inconditionnel de type GOTO à éviter**label1:**

```
instr1
instr2
goto label1
```

**label2:**

```
instr3
```

**4.1.2 Le saut conditionnel**

Un saut conditionnel permet d'avoir deux chemins différents en fonction du résultat d'une condition, il s'agit de la structure « IF THEN ELSE ». On va commencer par sa forme la plus simple:

**4.1.3 La forme « if then »**

Cette première version permet d'exécuter une partie de code lorsqu'une condition est vérifiée avant de poursuivre la suite du programme:

Voici le diagramme de flux correspondant:

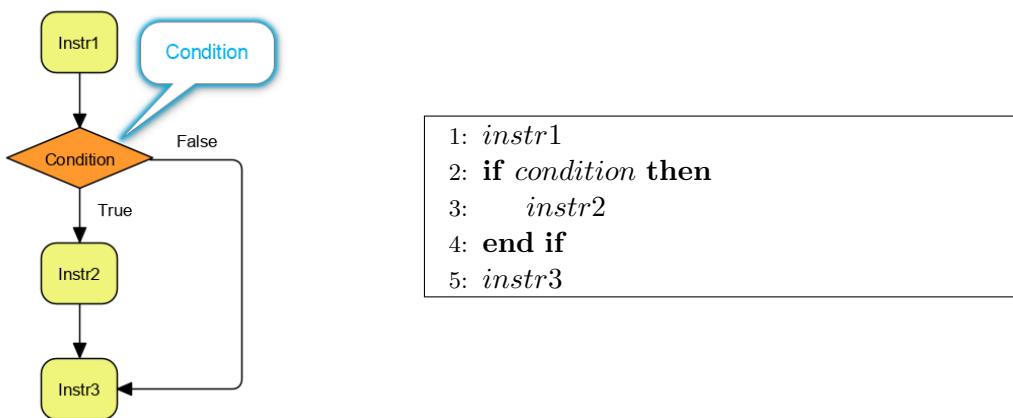


TABLEAU 4.1 – Structure IF THEN

Voici les représentations du « IF » en Java et en Python. Le tableau suivant montre les distinctions entre ces 2 langages:

	Python	Java
condition	<b>if</b> condition : après la condition il faut mettre « : »	<b>if</b> (condition) la condition doit être entre parenthèses
bloc à exécuter	les instructions doivent être indentées par des espaces	si le code contient plus d'une instruction, il faut les entourer par des accolades {}

---

**Algorithme 4.2** Rupture de séquence: « if then »
 

---



```
instr1
if condition:
    instr2
instr3
```

---



```
instr1;
if (condition) instr2;
instr3;
```

---

#### 4.1.4 La forme « if then else »

La clause « *else* » va permettre d'avoir deux chemins différents en fonction du résultat de la condition, avant de poursuivre la suite du programme.

Voici le diagramme de flux correspondant:

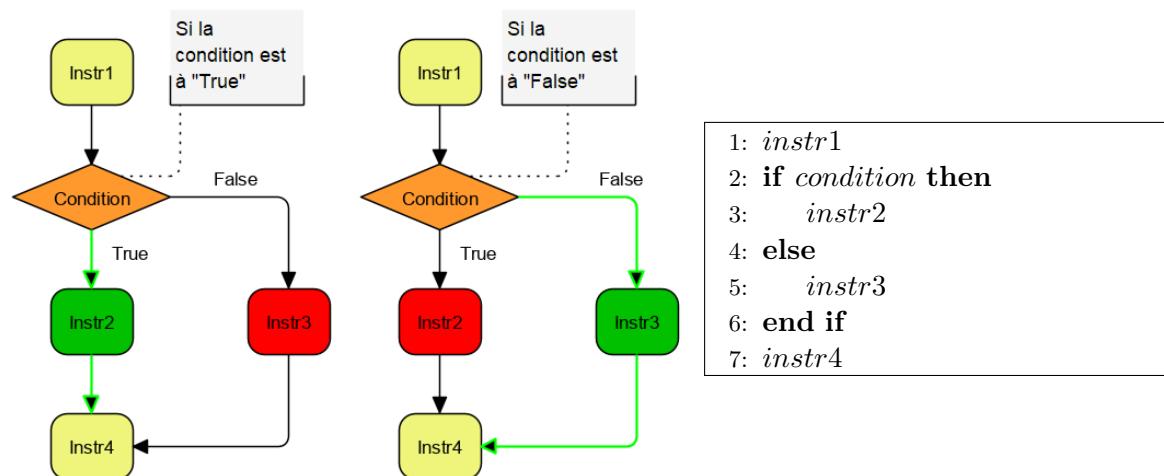


TABLEAU 4.2 – Structure IF THEN

Voici les représentations en Python et en Java

**Algorithme 4.3** Rupture de séquence: « *if then else* »

# Toujours bien respecter l'indentation, il est cependant possible  
 ↳ d'écrire l'instruction sur la même ligne que les deux points.



```
instr1
if condition:
    instr2
else :
    instr3
instr4
```

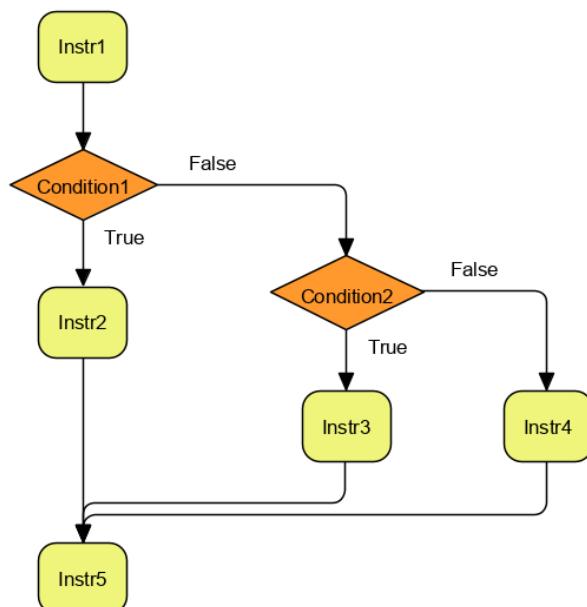
//L'indentation n'est pas obligatoire mais vivement recommandée  
 ↳ pour la lisibilité du code



```
instr1;
if (condition)
    instr2;
else
    instr3;
instr4;
```

**4.1.5 Les « if » imbriqués**

Il est assez fréquent d'avoir plusieurs tests imbriqués les uns dans les autres. Lorsque le test porte sur une même variable, il existe une forme de test plus appropriée, celle du « *selon que* » que nous aborderons plus loin 4.1.6. Cependant lorsque la structure du « *selon que* » n'est pas appropriée ou si le langage de la possède pas, il faudra imbriquer plusieurs structures « *if* » en faisant très attention de bien faire correspondre le « *else* » au bon « *if* ».



```

instr1
if condition1 then
    instr2
else
    if condition2 then
        instr3
    else
        instr4
    end if
end if
instr5
  
```

TABLEAU 4.3 – Structure avec des « *if* » imbriqués

Python	Java
<pre>instr1 if condition1 : instr2 elif condition2: instr3 else: instr4 instr5</pre> <ul style="list-style-type: none"> <li>☛ Python possède une clause « elif ».</li> <li>☛ Lorsqu'il n'y a qu'une instruction, on peut la mettre après les deux points.</li> </ul>	<pre>instr1; if (condition1)instr2; else if (condition2) instr3; else instr4; instr5;</pre>

---

**Algorithme 4.4** Rupture de séquence avec des « if » imbriqués

---

 # Toujours bien respecter l'indentation  
 instr1  
 if condition1:  
     instr2  
 elif condition2 :  
     instr3  
 else:  
     instr4  
 instr5

---

 //L'indentation n'est pas obligatoire mais vivement recommandée  
 → pour la lisibilité du code  
 instr1;  
 if (condition1)  
     instr2;  
 else if (condition2)  
     instr3;  
 else  
     instr4;  
 instr5;

---


**Il faut être certain que le ELSE se rapporte au bon IF**

Prenons l'exemple où deux « if » imbriqués sont suivis par un seul « else ». Comment peut-on indiquer que le « else » se rapporte au premier « if »? En Python c'est l'indentation qui le déterminera alors qu'en Java il faudra des accolades pour clôturer le deuxième « if ». Ce cas est présenté sur le schéma ci-dessous.

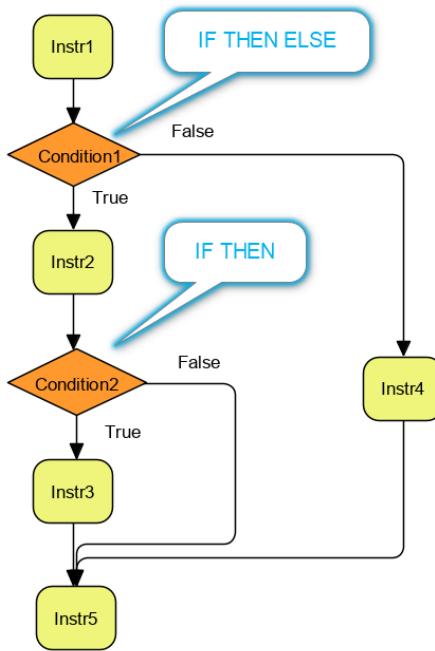


IMAGE 4.1 – Cas avec 2 « if » et un seul « else »

On constate sur le graphique que le premier test contient un « else » alors que le second test n'est possède pas. Il s'agit donc ici de clôturer le deuxième test en jouant avec l'indentation pour le langage Python et avec des accolades pour le langage Java.

---

#### Algorithme 4.5 Rupture de séquence avec des « if » imbriqués

---

#Toujours bien respecter l'indentation

```

instr1
if condition1:
    instr2
    if condition2 :
        instr3
else: #Le else est indenté au niveau du 1er IF
    instr4
instr5

```

---

//L'indentation n'est pas obligatoire mais vivement recommandée  
→ pour la lisibilité du code

```

instr1;
if (condition1)
    {instr2;
        if (condition2) //le IF est enveloppé par 2 accolades
            instr3;
    }
else instr4;
instr5;

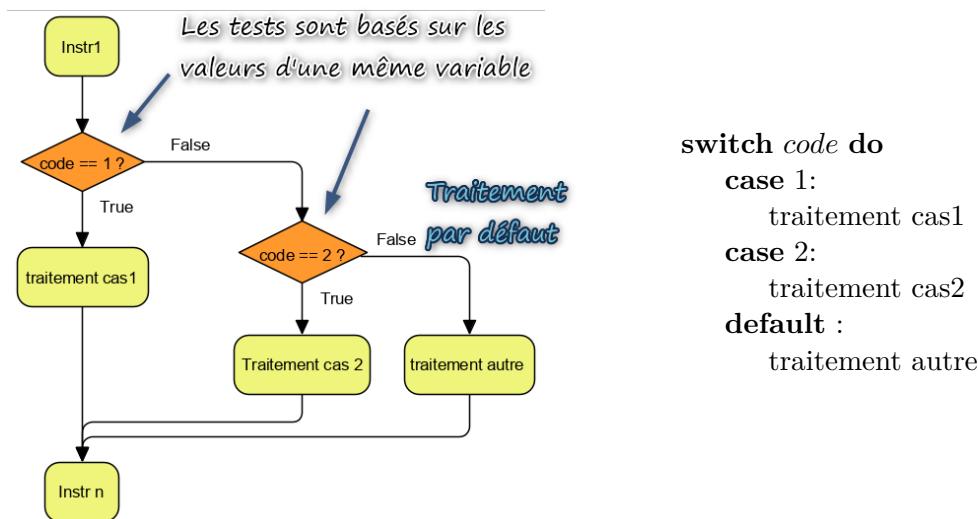
```

---

#### 4.1.6 Les sauts imbriqués « selon que »

Lorsqu'on a besoin d'exécuter un traitement différent en fonction de l'état d'une variable, on peut bien entendu utiliser des « if » imbriqués mais certains langages possèdent l'instruction « *switch* » ou

`match` » appelée aussi le « *selon que* » ou en anglais le « *case of* » qui permet de simplifier l'écriture.



```

switch code do
    case 1:
        traitement cas1
    case 2:
        traitement cas2
    default :
        traitement autre

```

TABLEAU 4.4 – Structure du « selon que »

Python possède depuis seulement sa version 3.10, une forme de « selon que » avec l'instruction « `match` » dont voici un exemple qui affiche le jour de la semaine en fonction d'un code allant de 1 à 7:

#### Algorithm 4.6 Version du « selon que » en Python

```

def getJour(jour : int) -> str:
    match jour:
        case 1:
            res = 'Lundi'
        case 2:
            res = 'Mardi'
        case 3:
            res = 'Mercredi'
        case 4:
            res = 'Jeudi'
        case 5:
            res = 'Vendredi'
        case 6 | 7:
            res = 'Weekend'
        case _:
            res = 'Pas un jour de la semaine'
    return res
print(getJour(2)) # Affichera: Mardi

```

 **Remarques sur l'instruction "match" de Python**

Un « case » peut contenir plusieurs instructions et ne nécessite pas de break. Il est aussi possible de faire une sélection sur base d'une expression régulière à la place d'une valeur immédiate.

Java propose plusieurs formes du « *switch* », dans la version la plus courante, chaque cas doit contenir une instruction « *break* » pour sortir du *switch*. Dans la version 17 de Java, deux autres formes plus synthétiques ont été rajoutées. Commençons par l'écriture la plus ancienne:

---

**Algorithme 4.7** Version1 du *switch* en Java
 

---



```

1 // "opération" est une variable du type énuméré "Operation"
2 switch (opération) {
3     case CREATE, INSERT:
4         System.out.println("Traitement Ajout");
5         break; //sort du switch
6     case UPDATE:
7         System.out.println("Traitement Mise à jour");
8         break; //sort du switch
9     case DELETE:
10        System.out.println("Traitement suppression");
11        break; //sort du switch
12    default:
13        System.out.println("Traitement autre");
14 }
```

---

Dans cet exemple, imaginons que la variable « *opération* » contienne « *Operation.UPDATE* », alors dans ce cas, seules les lignes 7 et 8 seraient exécutées par contre sans le *break* à la ligne 8, les instructions 10 et 11 seraient également exécutées.

La version suivante a été introduite avec Java 17, elle simplifie la notation en ne devant plus rajouter les « *break* ».

---

**Algorithme 4.8** Version2 du *switch* en Java
 

---



```

// "opération" est une variable du type énuméré "Operation"
switch (opération) {
    case CREATE, INSERT -> System.out.println("Traitement Ajout");
    case UPDATE -> System.out.println("Traitement Mise à jour");
    case DELETE -> System.out.println("Traitement suppression");
    default -> System.out.println("Traitement autre");
}
```

---

Lorsqu'on utilise un « *switch* » pour retourner une donnée en fonction de l'état d'une variable, on peut utiliser une 3<sup>ème</sup> forme du « *switch* », également introduite avec version 17 de Java:

---

**Algorithme 4.9** Version3 du *switch* en Java
 

---



```

// "devise" recevra la devise du pays"
String devise= switch (pays) {
    case BELGIQUE, FRANCE -> {yield "Euro";}
    case USA -> {yield "Dollar";}
    case GB -> {yield "Livre sterling";}
    default -> {yield "ToDo";}
}
```

---

Ici le *switch* renvoie la valeur indiquée après l'instruction « *yield* », donc si le pays est « USA » la devise sera « Dollar ».

## 4.2 Les différents types de boucles

Il arrive assez souvent dans un programme de devoir répéter plusieurs fois des instructions jusqu'à ce qu'une condition soit vérifiée ou pour un nombre de fois connu au départ. Une boucle n'est en fait rien d'autre qu'une rupture de séquence avec un retour en arrière, on pourrait envisager un simple « if » avec un saut inconditionnel qui renvoi en arrière, cependant cette structure n'est pas considérée comme claire et bien structurée (programmation Goto), surtout qu'il existe des alternatives beaucoup plus propres dans les langages de haut niveau. D'ailleurs les instructions de sauts inconditionnels n'existent ni en Java ni en Python.

Il existe cependant les boucles conditionnelles « **tant-que** » et « **répéter jusqu'à ce que** » et d'autre part les boucles « **pour** » qui permettent de parcourir une collection d'éléments ou de faire varier une variable un nombre de fois défini au départ. Commençons par les structures « **tant-que** ».

### 4.2.1 La boucle « tant-que »

On retrouve principalement 2 formes de boucle « **tant-que** », celle avec un test avant d'entrer dans la boucle et celle en fin de boucle. La condition est une condition pour rester dans la boucle.

Voici l'organigramme correspondant:

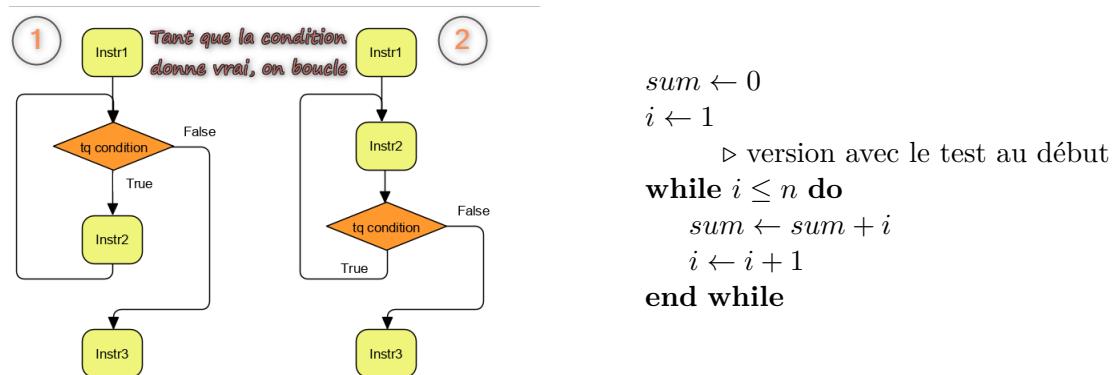
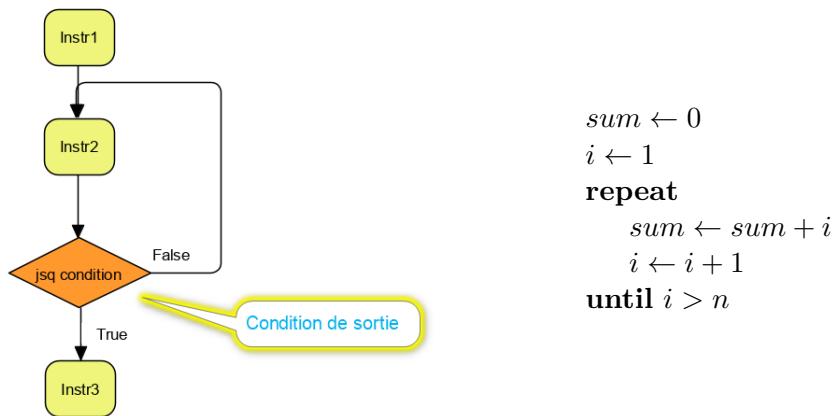


TABLEAU 4.5 – Structure d'une boucle « *tant que* »

Dans l'organigramme, j'ai présenté sur la gauche ①, une version avec le test en début de boucle, il s'agit de la version classique du « *tant que* » que l'on retrouve en pseudo-code, en *Python* comme en *Java*. Cependant la version ② avec le test en fin, existe en *Java* avec sa boucle « **do while** » mais pas en *Python* alors qu'en pseudo-code, on utilise une boucle « **répéter jusqu'à ce que** » où la condition est une condition de sortie de boucle (l'inverse de la condition du tant-que).

Regardons la boucle « **répéter jusqu'à ce que** » en pseudo-code même si elle n'existe pas en Java et Python:

TABLEAU 4.6 – Structure d'une boucle « *répéter jusqu'à ce que* »

Regardons maintenant ce que Java propose comme boucle « tant-que ». Dans cette première version, celle du « *while* », le test est vérifié avant d'entrer dans la boucle.

---

**Algorithme 4.10** La boucle « *while* » en Java
 

---

 *//Exemple qui montre les pertes avec les nombres réels*  
`double r=1.0;  
int cpt= 0;  
while (r>0){//Test pour rester dans la boucle  
r=r-0.2;  
cpt++;  
}  
System.out.println("r= "+r+" cpt= "+cpt );`

---

Dans la deuxième version, celle du « *do while* », la condition est testée en fin de boucle ainsi on rentrera toujours au moins une fois dans la boucle.

---

**Algorithme 4.11** La boucle « *do while* » en Java
 

---

 *//Exemple qui montre les pertes avec les nombres réels*  
`double r=1.0;  
int cpt= 0;  
//ici on rentre toujours au moins une fois dans la boucle  
do {  
r=r-0.2;  
cpt++;  
}while (r>=0);//test pour rester dans la boucle  
System.out.println("r= "+r+" cpt= "+cpt );`

---

Python de son côté ne propose qu'une seule version de la boucle « *while* », celle avec le test au début:

**Algorithme 4.12** La boucle « *while* » en Python

---

```
//Exemple qui montre les pertes avec les nombres réels
r=1.0
cpt= 0
python while r>0:#Test pour rester dans la boucle
    r=r-0.2
    cpt+=1
    print(f'r= {r} cpt= {cpt}')
```

---

**4.2.2 Exemple**

Je vais prendre un exemple classique pour illustrer la boucle « *tant-que* » en implémentant la méthode d'Euclide pour calculer le plus grand commun diviseur entre 2 nombres ou plus communément appelé le PGCD .

Hypothèse de départ : a et b sont 2 nombres entiers positifs

$pgcd(a, b) = pgcd(b, a \bmod b)$  et lorsque le reste ( $a \bmod b$ ) sera égal à 0, on obtiendra  $pgcd(x, 0)$  ou « x » sera le PGCD.

Prenons 2 nombres et regardons l'évolution de a et b :

a	b
15	9
9	6
6	3
3 (PGCD)	0 (Condition d'arrêt)

On peut facilement détecter une boucle qui s'arrête lorsque b sera à 0 avec le *pgcd* dans la variable « a ».

Voici l'écriture de cet algorithme sous forme d'une fonction codée en Python suivi d'une version en Java.

**Algorithme 4.13** Calcul du *pgcd* en Python

---

```
def pgcd(a: int,b: int)->int:
    '''Calcule le pgcd entre 2 nombres entiers positifs'''
    assert a>=0 and b>=0,"Les nombres doivent être des entiers
    → positifs"
    while b!=0:
        a, b = b, a%b
    return a
```

---

Voici la version écrite en Java. Elle nécessite une variable en plus pour calculer le reste.

---

**Algorithme 4.14** Calcul du *pgcd* en Java
 

---

```
//Calcul le PGCD selon l'algorithme d'Euclide
public static int pgcd(int a,int b){
  assert a>=0 && b>=0: "Les nombres doivent être des entiers
    → positifs";
  int reste;
  while (b!=0){
    reste=a%b;
    a=b;
    b=reste;
  }
  return a;
}
```



### 4.2.3 La boucle « pour »

Lorsqu'on connaît le nombre de passages que l'on doit effectuer dans une boucle, on opte pour l'instruction « *pour* ».

#### La boucle « pour » pour faire varier une variable d'indice

La première version de la boucle « *pour* » permet de faire varier une variable d'une *valeur\_1* vers une *valeur\_2* selon un *pas* généralement paramétrable. Commençons par voir sa structure en pseudocode suivie par sa représentation en Java. Cette dernière est très courante et similaire dans beaucoup de langages à l'exception de Python.

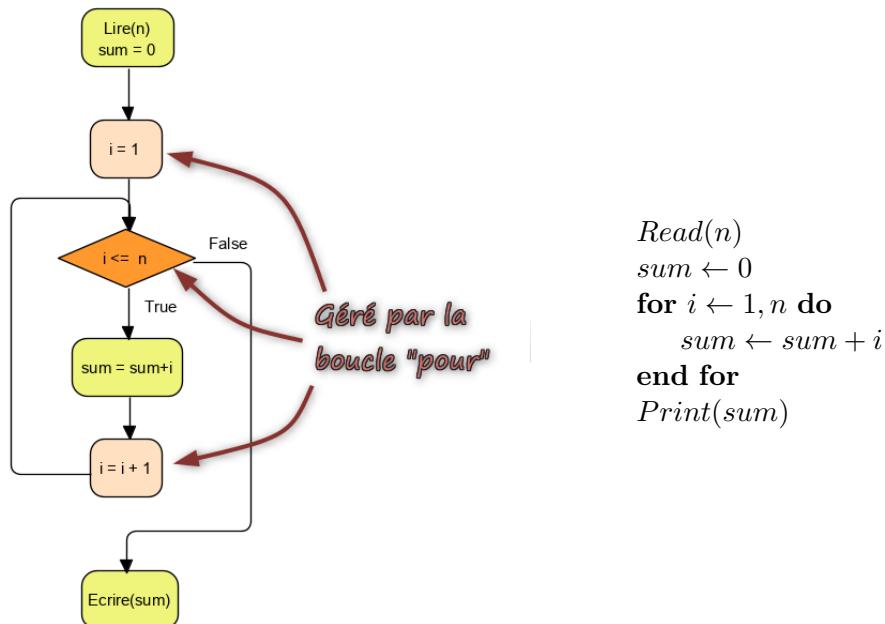


TABLEAU 4.7 – Structure classique d'une boucle « *pour* »

Commençons par regarder l'écriture de la boucle « *pour* » en Java car celle-ci est identique dans plusieurs langages:

Déclaration et initialisation d'une variable d'indice i à 1

Condition d'arrêt

Opération exécutée après chaque passage dans la boucle.  
Ici incrémenter i de 1

```
for (int i=1 ; i<=n ; i++)
    System.out.println(i);
```

Cette boucle fera varier i de 1 à n  
si n=0 on ne rentrera pas dans la boucle  
si n=1 on rentrera 1 fois dans la boucle



### Remarque

Il est déconseillé de modifier au sein du corps de la boucle, les variables qui peuvent influencer le nombre de passages dans la boucle, même si c'est autorisé dans certains langages. Dans l'exemple ci-dessus, on pourrait modifier « i » et « n » dans le code de la boucle mais c'est fortement déconseillé!

```
for (int i=1;i<n;i++) {
    System.out.println(i);
    n=n-1; // A EVITER
    i=i+2; //A EVITER
}
```

La variable « i » est une variable déclarée dans la boucle ainsi elle n'existera pas en dehors de celle-ci. Il est cependant possible de déclarer cette variable en dehors de la boucle mais ce pas une pratique courante.

```
int i;
for (i=1;i<5;i++)
    System.out.println(i); //Affichera 1 2 3 4 sur 4 lignes
System.out.println(i);//Affichera 5
```

Voici un exemple pratique qui calcule la factorielle de n. Sachant que :

$$\begin{cases} 0! = 1 \\ n! = 1 * 2 * \dots * (n - 1) * n \end{cases}$$

---

#### Algorithme 4.15 Calcul de n! en Java



```
//Calcul de la factorielle de n
public static long fact(int n ){
    assert n>=0 && n<21 : "Doit être en entier positif entre 0 et 20";
    long res=1;
    for(int i=2;i<=n;i++)
        res*=i;
    return res;
}
```

---

Python propose une boucle « pour » assez différente de la structure classique. Regardons d'abord les quelques exemples suivants:

**Algorithme 4.16** Boucle « pour » en Python

```
python    for i in range(4):
            print(i,end= ' ') # Affichera: 0 1 2 3
        for i in range(1,4):
            print(i,end= ' ') # Affichera: 1 2 3
        for i in range(1,10,2):
            print(i,end= ' ') # Affichera: 1 3 5 7 9
```

Regardons plus précisément le fonctionnement de cette boucle « pour », on constate la présence de la fonction intégrée « `range` » qui contient 3 paramètres:

1. la valeur initiale (par défaut à 0)
2. la valeur de fin obligatoire (valeur non comprise)
3. le pas (par défaut à 1)

Seul le 2<sup>ème</sup> paramètre est obligatoire, il désigne la valeur maximum (non comprise). Ces 3 paramètres peuvent être des expressions, ils seront évalués et fournis à la fonction intégrée « `range` », ainsi il ne sera plus possible de les modifier au sein de la boucle.

Donc la boucle « `for` » va initialiser une variable, « `i` » dans l'exemple, à la valeur initiale de la fonction « `range` » et lors de chaque passage la fonction `range` fournira la nouvelle valeur de « `i` » jusqu'à la dernière valeur avant la borne de fin. Ainsi

```
for i in range(4):
    print(i, end= ' ') # Affichera: 0 1 2 3
print(f' i= {i}') #Affichera: i= 3
```

Voici l'algorithme de la factorielle réécrite en Python:

**Algorithme 4.17** Calcul de « `n!` » en Python

```
python    def factorielle(n):
            ''' Calcule la factorielle de n '''
            assert type(n)== int and n>=0, "n doit être un entier positif"
            res=1
            for i in range(2,n+1):
                res = res * i
            return res # Retourne n!
```

Bien entendu, on pourrait aussi réécrire l'algorithme en éliminant la variable « `i` » et plutôt utiliser la variable « `n` » que l'on décrémenterai par pas de -1:

**Algorithme 4.18** Calcul de « n! » version2

```
def factorielle(n):
    ''' Calcule la factorielle de n '''
    assert type(n)== int and n>=0, "n doit être un entier positif"
    res=1
    #Range va fixer la valeur initiale à n
    for n in range(n,1,-1): #n va aller de n à 2
        res = res * n
    return res # Retourne n!
```



```
public static long factorielle(int n) {
    assert n >= 0 && n<21 : "Le nombre doit être
    ↪ positif et inférieur à 21";
    long res = 1;
    //n est déjà initialisé ainsi la 1ère partie du for
    ↪ est vide. On avance ici par pas de -1
    for (; n>1; n--)
        res = res * n;
    return res;
}
```

**Noms des variables d'indice**

Dans une boucle « pour », Il est assez fréquent d'utiliser des noms de variable d'indice comme `i`, `j`, `k`, ...

**La boucle « pour » pour parcourir un ensemble**

Java et Python possède une boucle du style « `for each` » qui permet de parcourir chaque élément d'une collection, d'une liste, d'un ensemble,...

Nous n'avons pas encore aborder les structures de liste, d'ensemble, de dictionnaire, ni la notion d'objet mais nous allons malgré tout regarder le fonctionnement de cette boucle qui est assez intuitif.

Commençons par un exemple en Java qui affiche la somme d'une liste de nombres premiers:

**Algorithme 4.19** Boucle « `for each` » en Java

```
//Liste avec les 8 premiers nbrs premiers
List<Integer> listePremier= List.of(1,2,3,5,7,11,13,17);
long sum=0;
//La boucle va faire 8 passages
//La variable d'indice "i" est du même type que les éléments de la
↪ liste
for (Integer i: listePremier) {
    sum = sum + i;
    System.out.println("somme= "+sum);
}
```

**4.3 L'appel et le retour d'une fonction**

Lorsqu'on appelle une fonction, on rompt la séquence en donnant le contrôle au code de celle-ci et à la fin de la fonction, on retourne au programme appelant en renvoyant ou non des données. L'appel

de la fonction consistera à sauter sur la première instruction de celle-ci après avoir rajouté sur la pile l'adresse de retour, les éventuels paramètres effectifs ainsi que les variables locales utilisées dans le code de la fonction. Ces aspects seront étudiés en détail dans la partie consacrée aux procédures et fonctions.

## 4.4 La rupture de séquence suite au déclenchement d'une exception

Une rupture de séquence peut également être due au déclenchement d'une exception. On peut distinguer deux grandes catégories d'exception, celles que l'on peut gérer et celles qui vont entraîner systématiquement un arrêt du programme. Dans le premier cas, l'erreur va déclencher une exception qui dévierà la suite du programme vers un bloc de code dédié à la traiter.

Prenons comme exemple l'ouverture d'un fichier, si le fichier n'est pas présent, une exception sera déclenchée pour signaler ce problème; cette erreur peut très bien être corrigée d'où l'intérêt de traiter l'exception pour informer l'utilisateur. Le traitement de ce type d'erreur sera détaillé dans la partie consacrée aux fichiers (12).

Voici un autre exemple d'erreur rattrapable, lorsqu'on désire convertir une chaîne de caractères en une valeur numérique (*parse* un nombre), si l'utilisateur encode une chaîne erronée, elle ne pourra pas être *parsed* et déclenchera une exception. Si l'on attrape cette exception, on peut très bien informer l'utilisateur que la valeur entrée n'est pas valable et qu'il doit réintroduire une autre valeur.

La gestion des exceptions sera abordée plus loin dans un chapitre consacré aux exceptions (13)

# Chapitre 5

## Les opérateurs

### 5.1 Opérateurs arithmétiques

Opérations	Python	Java	priorité
addition	<code>x+y</code>	<code>x+y</code>	3
soustraction	<code>x-y</code>	<code>x-y</code>	3
multiplication	<code>x * y</code>	<code>x * y</code>	2
division réelle	<code>x/y</code>	réel x/ réel y	2
division entière	<code>x//y</code>	entier x / entier y	2
modulo	<code>x%y</code>	<code>x%y</code>	2
puissance	<code>x**y</code>	<code>Math.pow(x,y)</code>	1
opposé (unaire)	<code>-x</code>	<code>- x</code>	1

TABLEAU 5.1 – Opérateurs Arithmétiques

#### 5.1.1 Ordre de priorité dans l'évaluation d'une expression

Les opérateurs unaires vont être les premiers à être évalués, ensuite se sera les opérateurs de niveau 2 (multiplications, divisions,...) suivi par les opérateurs de niveau 3 (additions et soustractions):

```
>>> -5+2*-4
-13
>>> ((-5)+(2*(-4)))
-13
```

Lorsque les opérateurs binaires possèdent le même niveau de priorité l'évaluation se fera de gauche à droite :

```
>>> 6*5//2          6*5 => 30 et 30 //2 ==> 15
15
>>> 5//2*6          5//2 => 2 et 2 * 6 ==> 12
12
```

Lorsque les opérateurs unaires se suivent l'évaluation de fera de droit à gauche:

```
>>> --5          identique à (-(-5))
5
```

## 5.2 Type Casting

Quand une donnée est stockée en mémoire, on lui associe une type qui permet de savoir comment interpréter ses bits. Par défaut le type est défini lors de création de la variable ou en Python lors de son assignation. Cependant dans plusieurs situations, on a besoin de traiter l'information avec un autre type ou de changer le type initial, pour ce faire on effectue un typecasting. Bien entendu, on ne peut pas appliquer un nouveau type à une variable si celle-ci n'est pas « compatible ».

On utilise souvent le *typeCasting* en programmation orientée objet mais il est un peu tôt pour bien comprendre son utilité. Cependant voici des exemples simples en Python et en Java qui utilisent le *typeCasting* sur des types élémentaires:

**Syntaxe en Python:** *le nouveau type (expression)*

**Syntaxe en Java:** *(le nouveau type) expression*

---

### Algorithme 5.1 Type Casting

---

```
r = 3.6 # r est de la classe "float"
i = int(r) # i est de la classe "int" et contiendra 3 grâce au
           → typeCasting
python
#La lecture de l'input génère une chaîne de caractères qui sera
→ convertie en entier avec un TypeCasting
n = int(input('Entrez un nombre'))

double r=3.61
//Attention au débordement si r ne peut pas être transformé en int
int a= (int) r; //casting vers un entier: a sera égal à 3
```

Conversion automatique de : byte -> short -> int -> long -> float -> double  
 Conversion manuelle pour : double -> float -> long -> int -> short -> byte

---

## 5.3 Manipulation des nombres réels

### 5.3.1 Les types réels dans les langages de programmation

Pour coder des variables réelles, les langages utilisent principalement les types « *float* » et « *double* », ils respectent la norme *IEEE754* IEEE754

Remarque: ces types possèdent des codes binaires spécifiques pour représenter les nombres mathématiques  $\pm\infty$  et un résultat *indéterminé*.

	Types	Taille
	<i>float / Float</i>	Réel sur 32 bits
	<i>double / Double</i>	Réel sur 64 bits

	Types	Taille
	La classe « <i>float</i> » utilise un type <i>double</i>	Réel sur 64 bits

### 5.3.2 La codification des nombres réels en base 2

Le manipulation des nombres réels est assez complexe et nécessite d'être vigilant principalement lors des tests d'égalité. Commençons par un exemple où je retire du nombre réel « 1.0 » cinq fois le nombre réel « 0.2 », surprise le résultat de sera pas égal à 0:

```
>>> 1.0 - 0.2 - 0.2 - 0.2 - 0.2 - 0.2
```

```
5.551115123125783e-17
```

Le résultat n'est pas loin de 0 mais n'est pas égal à zéro!

Voici l'explication:

La plupart des nombres réels en base 10 ne peuvent pas s'écrire sans erreur sur un nombre fini de bits prenons  $0.2_{10}$  et essayons de le convertir en base 2:

Multiplie successivement par 2	Bits après la virgule
$0.2 * 2 = 0.4$	0
$0.4 * 2 = 0.8$	0
$0.8 * 2 = 1.6$	1
$0.6 * 2 = 1.2$	1

$$\Rightarrow 0.2_{10} = 0.00110011 \dots 0011$$

Comme on peut le constater, « 0.2 » nécessite une infinité de bits pour être représenté sans erreur, ce sera d'ailleurs le cas aussi pour les nombres « 0.1 », « 0.4 », « 0.6 », « 0.7 », « 0.8 » et « 0.9 ».

Étant donné que l'on ne peut pas coder une infinité de bits, il y aura une erreur de précision qui nous donnera en fonction qu'un arrondi soit appliqué, soit une valeur en dessous (sans arrondi) soit une valeur au dessus (avec un arrondi) soit une valeur exacte si le nombre peut être codé sans erreur.

### 5.3.3 Test d'égalité avec des nombres réels:

Il ne faudra jamais tester si deux expressions réelles sont identiques mais si deux expressions réelles sont proches l'un de l'autre. Il faut se fixer une précision et tester si la différence des 2 expressions réelles est inférieur en valeur absolue à notre précision:

$$|a - b| < \epsilon$$

Voici un extrait d'un code qui retourne le nombre de racine d'une équation du 2<sup>ème</sup> degré en évaluant delta :  $\Delta = b^2 - 4 * a * c$

```
public byte getNbSol() {
    if (Math.abs(delta) < 0.0000001)
        return 1;
    if (delta > 0)
        return 2;
    return 0;
}
```

*notre précision*

*nombre réel:  $b^2 - 4*a*c$*

### 5.3.4 Expression comprenant des nombres mélangeant des types différents

Prenons un exemple en Java où une expression contient des variables (ou valeurs immédiates) entières et réelles. On peut constater un résultat inattendu pour certains:

```
int i = 5;
double r1 = 2.5 + i / 2;    //r1 contiendra 4.5
double r2 = 2.5 + i / 2.0; //r2 contiendra 5.0
```

Pour « r1 », le premier membre de l'addition est évalué en réel et le deuxième membre comme une expression entière d'où le résultat.

Pour « r2 », comme le deuxième membre contient une valeur réelle dans l'expression, toute l'expression sera traitée en réel.

On parle de promouvoir une variable d'un type vers un autre type de taille supérieure (ici d'entier en réel).

### 5.3.5 Librairies mathématiques

Chaque langage propose des librairies mathématiques que l'on peut importer et utiliser directement. Je n'énumère ici qu'une librairie classique pour chaque langage:

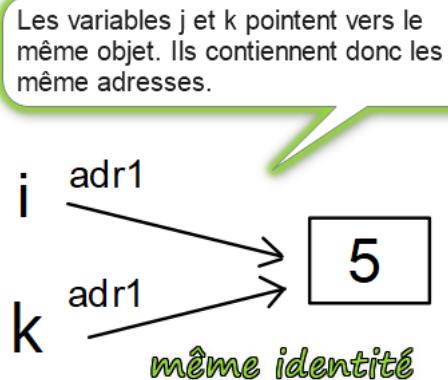
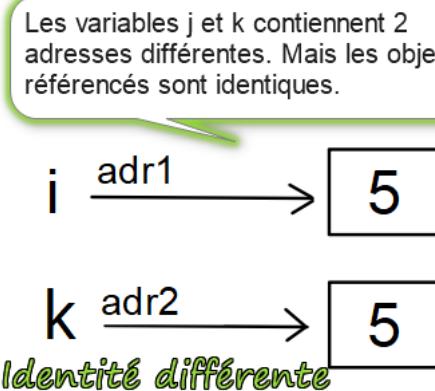
Langage	Librairie/ Module	Exemple
Java	<u>La classe: <code>java.lang.Math</code></u>	<code>Math.sin(Math.PI / 3)</code>
Python	<u>Le module math: <code>import math</code></u>	<code>math.sin(math.pi / 3)</code>

En Java la classe « *Math* » fait partie du package principal « *java.lang* » et de ce fait, elle ne nécessite pas d'importation.

Pour Python, vous devrez importer le module « *math* » en faisant « `import math` », une autre librairie mathématique fortement utilisée en Python pour le calcul vectoriel est « *numpy* ».

## 5.4 Opérateurs de comparaison

Pour bien comprendre le fonctionnement de ces opérateurs, il ne faut pas oublier qu'en Python tout est objet, ce qui signifie qu'une variable contient toujours l'adresse d'un objet, prenons le cas d'une variable entière `i=5`, « *i* » contiendra l'adresse d'une zone mémoire contenant l'information « 5 » et non pas la valeur 5. Par contre en Java, une variable de type primitif contient directement l'information ainsi `int i=5;`, après l'assignation, « *i* » contiendra bien la valeur 5. Toutes les variables d'un type objet contiendront, comme en Python, des référents vers les objets associés. Ainsi il faudra bien distinguer la notion d'égalité entre 2 variables et d'égalité entre 2 objets.



Ainsi, lorsqu'on désire tester l'égalité entre 2 objets, il faudra appeler une méthode (fonction) de l'objet qui devra comparer les contenus des objets et ainsi déterminer s'ils sont identiques ou non.

- Pour savoir si 2 référents pointent vers le même objet, il faudra utiliser les opérateurs:

**Java** il faut utiliser l'opérateur « `==` »: `i == k`

**Python** il faut utiliser l'opérateur « `is` »: `i is k`

- Pour déterminer si 2 objets sont identiques par leur contenu, il faudra utiliser les opérateurs:

**Java** utiliser la fonction « `equals` »: `i.equals(k)`

**Python** utiliser l'opérateur « `==` » qui indirectement appelle la méthode: « `i.__eq__(k)` »

Dans l'exemple suivant, on peut constater la différence entre l'égalité des référents et celle des contenus des objets. Au départ, j'initialise 2 variables `t1` et `t2` avec le même texte, on constate que Java ne duplique pas l'objet mais ceci est dû à une astuce interne à la JVM:

```
jshell> String t1="hello Didier";
t1 ==> "hello Didier"

jshell> String t2="hello Didier";
t2 ==> "hello Didier"

jshell> t1==t2          Pour des raisons d'optimisation t1 et t2 pointent vers le
$3 ==> true             même objet

jshell> t1.equals(t2)
$4 ==> true
```

Pour poursuivre l'exemple, je modifie `t1` et `t2` en leurs concaténant "2", ainsi elles contiendront toujours le même texte. On peut constater sur l'image suivante que les deux objets contiennent bien le même texte mais plus les mêmes adresses:

```
jshell> t1=t1+2
t1 ==> "hello Didier2"
jshell> t2=t2+2
t2 ==> "hello Didier2"
jshell> t1==t2          t1 et t2 contiennent le même texte
$7 ==> false            On constate ici que t1 et t2 pointent vers des objets différents
jshell> t1.equals(t2)    alors que leurs contenus sont identiques
$8 ==> true
```

Opérations	 Python	 Java	priorité
Égal	<code>x == y</code> appelle la méthode <code>x __eq__ (y)</code>	(sur les types primitifs) <code>x == y</code> (sur les objets) <code>x.equals(y)</code>	4
Différent de	<code>x != y</code> ou <code>x &lt;&gt; y</code> appelle la méthode <code>x __ne__ (y)</code>	(sur les types primitifs) <code>x != y</code> (sur les objets) <code>! x.equals(y)</code>	4
Plus Petit	<code>x &lt; y</code>	<code>x &lt; y</code>	4
Plus Petit ou Égal	<code>x &lt;= y</code>	<code>x &lt;= y</code>	4
Plus Grand	<code>x &gt; y</code>	<code>x &gt; y</code>	4
Plus Grand ou Égal	<code>x &gt;= y</code>	<code>x &gt;= y</code>	4
Identité (adresses identiques)	<code>x is y</code>	<code>x == y</code>	4
Pas identique (adresses différentes)	<code>x is not y</code>	<code>x != y</code>	4

TABLEAU 5.2 – Opérateurs de comparaison

Pour illustrer la priorité des opérateurs de comparaison, j'ai repris l'exemple suivant qui montre que le test d'égalité se fera en dernier lieu, d'où mon choix d'attribuer un niveau de priorité 4:

`>>> -5+2*-4 == -13+0`



True      Priorité 4

## 5.5 Opérateurs logiques

Opérations	 Python	 Java	priorité
Et (produit logique)	<code>and</code>	<code>&amp;&amp;</code>	2
Ou (somme logique)	<code>or</code>	<code>  </code>	3
Non logique	<code>not</code>	<code>!</code>	1

TABLEAU 5.3 – Opérateurs de comparaison

Il ne faut pas oublier que le « `and` » est plus prioritaire qu'un « `or` » car l'un est un produit logique alors que l'autre est une somme logique.

Voici un exemple en Python qui montre l'ordre de priorité lors de l'évaluation de la condition:

`a>6 or b==5 and a<2`  
*Sera évalué de la façon suivante*  
`(a>6) or ((b==5) and (a<2))`

En Java, on obtiendrait le même résultat à l'exception que les opérateurs se noteraient différemment.

### 5.5.1 Évaluation d'une expression lorsque le résultat est évident:

Dans la plupart des langages, lorsque le résultat d'une expression est connu, le reste de l'expression ne sera plus évalué. Prenons l'expression suivante où lorsque la variable « *a* » est à 0, le reste de l'expression « *b/a* » ne sera jamais exécuté car comme le premier terme du « *and* » est faux, le résultat de l'expression est connu sans devoir évaluer la suite. C'est d'autant plus important ici car sinon, on aurait eu une erreur en calculant « *b/0* »! L'ordre dans l'expression a dans ce cas une importance capitale!

```
a=0
b=8
if a!=0 and b/a >5: print(a,b)
```

Cette division ne sera jamais effectuée lorsque a=0

En programmation orientée objet, il est assez fréquent de faire un test sur l'existence de l'objet avant d'appeler une méthode sur l'objet:

```
//Exemple en java
//on s'assure que le bic existe avant de regarder son niveau
if (bic != null && bic.getNiveau()<2)
    System.out.println("bic presque vide");
```

## 5.6 Opérateurs binaires

Opérations bits à bits	Python	Java	priorité
Et binaire	&	&	2
Ou binaire			3
Non binaire (complément à 1)	~	~	1
Ou exclusif	^	^	3
Décalage gauche	<<	<<	1
Décalage droit (non signé)		>> (injecte des 0 à gauche)	1
Décalage arithmétique droit (signé)	>> (injecte le signe à gauche)	>> (injecte le signe à gauche)	1

TABLEAU 5.4 – Opérateurs Binaires

**⚠ La priorité de ces opérateurs est moins prioritaire que celle des opérateurs arithmétiques (Java et Python)**

Dans l'exemple suivant, l'opérateur de décalage « << » est moins prioritaire que l'addition, ainsi les parenthèses sont obligatoires sinon il effectuera 1 décalé à gauche de (32+n) positions.

```
# n désigne un entier, l'addition sera prioritaire sur le décalage
n = (1<< 32) + n
```

En Python les entiers sont des objets et comme chaque opérateur est en réalité dévié vers un appel de méthode, il n'est pas possible d'avoir accès à la codification réelle en mémoire. Prenons l'exemple de « -1 » qui en complément à 2 sera codé par une suite de « 1 » sur 1 bits.

### 5.6.1 Fonctionnement et utilités des opérateurs binaires

Regardons d'abord la table de vérité de ces opérateurs avant de montrer leurs intérêts au niveau de la programmation.

a	b	a & b	a   b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

TABLEAU 5.5 – Table de vérité des opérateurs binaires « *and* », « *or* » et « *xor* »

#### Effectuer des opérations sur le contenu d'une variable:

Quand vous assignez une valeur dans une variable, tous les bits de celle-ci seront affectés. Si vous comptez définir ou analyser certains bits d'une variable, sans modifier et sans se baser sur l'état de ses autres bits, le seul moyen sera d'utiliser les opérateurs binaires. Il est cependant très fréquent de devoir créer ou analyser une suite de bits par exemple en décortiquant une trame réseau, en analysant un fichier binaire comme une image ou pour coder et décoder un fichier en UTF8,...

Les opérateurs binaires vont venir à notre rescousse en effectuant la plus part du temps une opération entre la variable et une suite de bits que l'on définit judicieusement, cette dernière s'appelle généralement le **masque**. Dans l'exemple suivant on désire savoir si le 2<sup>ème</sup>bit de n est à 1 en faisant une opération de « et logique » entre la variable « n » et un masque qui contient juste un « 1 » en 2<sup>ème</sup> position; lorsque le résultat est plus grand que 0 (ou égal au masque), cela signifie que la variable « n » possède bien un 1 pour son 2<sup>ème</sup> bit:

n:	01010010
masque:	00000010
<hr/>	
n & masque:	00000010 = masque $\Rightarrow$ le 2 <sup>ème</sup> bit de « n » est bien à 1

Voici un résumé des opérations qu'on peut effectuer à partir des opérateurs binaires:

Opérateur	Utilité	État du masque	Exemple
AND	Forcer des bits à 0	0 pour forcer à 0 1 pour ne rien changer	n: 01101010 masque: 11000111 n $\leftarrow$ n & masque: 01000010
	Extraire des bits d'une zone	1 pour la zone à extraire 0 ailleurs	n: 01101010 masque: 00111000 res $\leftarrow$ n & masque: 00101000
OR	Forcer des bits à 1	1 pour forcer à 1 0 pour ne rien changer	n: 01101010 masque: 00111000 n $\leftarrow$ n   masque: 01110101
	Écrire des bits dans une zone	(zone à recevoir à 0) 0 pour les bits à recopier x pour les bits à insérer	n: 01000010 masque: 00101000 n $\leftarrow$ n   masque: 01101010
XOR	Inverser certains bits	1 pour les bits à inverser 0 ailleurs	n: 01101010 masque: 00111000 n $\leftarrow$ n $\hat{\wedge}$ masque: 01010010
	Mélanger 2 données	mélange $\leftarrow$ d1 $\hat{\wedge}$ d2 $\Rightarrow$ d1 = d2 $\hat{\wedge}$ mélange d2 = d1 $\hat{\wedge}$ mélange	d1: 01011010 d2: 10011100 m $\leftarrow$ d1 $\hat{\wedge}$ d2 : 11000110

TABLEAU 5.6 – Utilités des opérateurs binaires « and », « or », « xor »

### Décalages pour effectuer des multiplications par $2^n$ :

Si vous multipliez  $123_{10}$  par  $10^2$  vous obtiendrez  $12300_{10}$ , ce qui revient à faire deux décalages vers la gauche. Ceci fonctionne pour toutes les bases ainsi  $1011_2 * 16 = 1011_2 * 2^4 \Rightarrow 10110000_2$ . On peut se poser la question à savoir « pourquoi ne pas faire une multiplication ? », tout simplement parce que la multiplication est beaucoup plus couteuse que les opérateurs binaires qui sont eux très rapides!

### Décalages pour effectuer des divisions par $2^n$ :

Lorsque vous devisez un nombre par sa base, cela revient à faire un décalage vers la droite d'une position, cependant la division pose plus de problèmes lorsqu'il s'agit d'un nombre codé en complément à 2.

Regardons un premier exemple avec un nombre en base 2 sans préciser de taille mémoire:

$$1010011,0 / 2^3 = 1010011,0_2 >> 3 \Rightarrow 1010,011_2 = 10,375_{10}$$

Maintenant si je prends le même exemple mais codé comme un entier positif sur 8 bits:

$10100011_2 / 2^3 = 10100011_2 >> 3 \Rightarrow 10100_2 = 20$ , ce qui revient à faire une division entière! On peut aussi constater que :

$$10100011_2 \& 00000111_2 = 00000011_2 = 3_{10}, \text{ qui correspond au reste de la division par } 2^3.$$

Le problème se complique lorsqu'on travaille avec des nombres signés codés en complément à 2, comme c'est le cas sur la plupart des ordinateurs. Prenons comme exemple « -6 » codé sur 8 bits en complément à 2:

$$-6_{10} \Rightarrow 11111010_2, \text{ sur 16 bits on aurait } 111111111111010_2,$$

on constate qu'on peut rajouter autant de « 1 » à gauche sans changer la valeur du nombre. Lorsque le nombre est positif, ce sont des zéros que l'on peut rajouter à gauche sans changer la valeur. Ainsi lorsqu'un nombre est codé en complément à 2, on peut répéter le signe (1/0) autant de fois que désirer sans changer la valeur du nombre.

Après avoir vu ces aspects, on comprend mieux pourquoi il existe 2 types de décalages vers la droite:

1. Décalages sur un nombre signé codé en complément à 2:

a) décalage arithmétique droit sur un nombre signé:

$-14_{10}$  sur 8 bits = 11110010<sub>2</sub>. Divisé par 8 donnera : 11110010<sub>2</sub> >> 3  $\Rightarrow$  11111110<sub>2</sub> =  $-2_{10}$

a) décalage classique droit sur un nombre signé:

$-14_{10}$  sur 8 bits = 11110010<sub>2</sub>. Divisé par 8 donnera : 11110010<sub>2</sub> >>> 3  $\Rightarrow$  00011110<sub>2</sub> =  $+30_{10}$

2. décalage classique avec les nombres non signés, on injecte des zéros à gauche:

$14_{10}$  sur 8 bits = 00001110<sub>2</sub> Divisé par 8 donnera : 00001110<sub>2</sub> >> 3  $\Rightarrow$  00000001<sub>2</sub> =  $1_{10}$



### Attention avec Python

Python encode les nombres entiers d'une manière différente de la plupart des langages, par exemple il n'existe pas de limite pour la taille d'un entier, il code cela comme une suite de bits. Cependant, il utilise bien la codification du complément à 2 pour les nombres signés!

En Python, il n'y a qu'un seul type de décalage, le décalage droit.

$-14_{10} >> 3 \Rightarrow -2_{10}$  c'est un décalage arithmétique droit comme en Java.

## 5.7 Opérateurs sur des caractères

Opérations	Python	Java
Concaténation	"ab"+ "cd"	"ab"+ "cd"
Répétition	"ab" * 2 ==> "abab"	
Code unicode d'un caractère	ord('A') ==> 65	(int) 'A' ==> 65
Caractère à partir de son code unicode	chr(65) ==> 'A'	(char) 65 ==> 'A'

TABLEAU 5.7 – Opérations sur les caractères

## 5.8 Opérateur ternaire

Vous connaissez probablement la fonction « SI » en Excel, elle permet de retourner une valeur différente en fonction du résultat d'une condition, ce sera le rôle de l'opérateur ternaire qui existe en Java comme en Python. Regardons les 2 notations assez différentes:

### Algorithme 5.2 Opérateur ternaire

 python	<code>#res recevra la valeur 1 si la condition est vraie sinon elle</code> <code>→ recevra la valeur2</code> <code>res = valeur1 if a &lt; b else valeur2</code>
	<code>//res recevra la valeur1 si a est plus petit que b sinon elle</code> <code>→ recevra la valeur2</code> <code>int res= a&lt;b ? valeur1:valeur2;</code>

## Chapitre

# 6

## Les structures à 1 ou plusieurs dimensions

Dans cette partie, nous allons nous intéresser aux structures à une ou plusieurs dimensions, en commençant par les vecteurs (structure à une dimension) suivi par les tableaux (structure à deux dimensions). Il sera bien entendu possible de définir des structures à plus de 2 dimensions car le principe restera le même.

Voici quelques règles spécifiques à ces structures:

- en Java une structure est en réalité une classe. Cependant elle ne possède pas de méthode propre à elle, juste les méthodes de la classe « *Object* » ainsi qu'un attribut non modifiable « *length* ».
- toutes les cellules d'un vecteur auront le même type ou un type compatible
- la taille d'un vecteur sera fixée lors de sa création
- les cellules seront accessibles via un indice qui commence à 0

Cette partie sera plus axée sur Java car il n'existe pas de type élémentaire identique en Python, cependant les listes Python(7.1) sont assez proches au niveau de la manipulation (création, accès, assignation), mais à coté de cela, elles possèdent plusieurs méthodes pour faciliter des opérations comme l'insertion d'un élément à l'intérieur d'une liste, opérations à gérer soi-même sur des *arrays* Java.

En Python vous retrouverez les notions de tuple, d'ensemble, de liste et de dictionnaire. En Java aussi, vous trouverez des classes pour gérer des listes, des ensembles et des dictionnaires. Cependant Java fonctionne d'une manière traditionnelle en utilisant un constructeur pour créer l'objet et en faisant appel à des méthodes pour y accéder, Python par contre facilite la syntaxe en utilisant des opérateurs classiques qui sont déviés sur des méthodes:

---

### Algorithme 6.1 Liste Python vs Liste Java

---

 python    `l=[10,20,30,40];  
l[0]=l[1]+l[2]; #=>l=[50,20,30,40]`

---

 Java    `List<Integer> l= new ArrayList<>(List.of(10,20,30,40));  
l.set(0,l.get(1)+l.get(2)); //=>l=[50,20,30,40]`

---

### 6.1 Les vecteurs

Un vecteur est une structure à une dimension qui possèdera plusieurs cases consécutives du même type. Prenons l'exemple d'un vecteur « V » d'entiers:

V	1	2	3	5	8	13
indices:	0	1	2	3	4	5

La taille du vecteur est de 6 entiers, chaque case est accessible via son indice ainsi:

$v[0] \Rightarrow 1, v[4] \Rightarrow 8$

Regardons maintenant comme créer et utiliser un vecteur:

### 6.1.1 Déclaration d'une variable de type vecteur:

Lors de la déclaration de la variable, on ne précise pas la taille du vecteur mais uniquement le type de ses éléments:

```
int[] v;
```

On constate la présence de 2 crochets « [ ] » après le type « int », le type peut très bien être un nom de classe comme

```
String[] s;
```

, d'ailleurs si vous regardez le paramètre d'une méthode « main », vous trouverez en entrée un vecteur de String qui permet de fournir au programme des données lors de son lancement:

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

Une variable de type vecteur contiendra en réalité l'adresse mémoire de l'endroit où se situe le vecteur!

### 6.1.2 Création du vecteur

Maintenant que l'on a défini une variable de type vecteur, on va pouvoir lui assigner un vecteur. Pour ce faire, il faudra demander de la mémoire dynamiquement et fournir l'adresse de cette zone dans la variable. Ici je vais créer un vecteur de 6 entiers qui seront initialisés par défaut à 0, comme il s'agit de 6 entiers de 32bits, un zone de mémoire de 24 bytes sera nécessaire.

```
//Création d'un vecteur de 6 entiers  
v = new int[6];
```

Lorsqu'un vecteur sera créé de cette façon, les cellules du vecteur seront initialisées par une valeur par défaut:

- ☛ types numériques ==> « 0 » pour les entiers et « 0.0 » pour les réels
- ☛ type booléen ==> « false »
- ☛ pour les classes ==> le pointeur « null »

#### Création d'un vecteur initialisé:

Il est possible d'initialiser un vecteur avec des données autres que celles par défaut en indiquant entre 2 accolades les valeurs du vecteur séparées par des virgules. Voici l'initialisation du vecteur « V »:

```
int[] v = {1,2,3,5,8,13};
```

Comme vous pouvez le constater sur ce dernier exemple, il est possible de déclarer la variable « v » et de l'initialiser en une seule ligne.

### Une variable vecteur est un pointeur:

Si j'affiche la variable « v », vous apercevrez une adresse, si vous assignez la variable « v » à une autre variable de même type, les deux variables pointeront sur le même vecteur. Il ne faudra pas oublier ceci lors du passage de paramètre dans une fonction!

---

#### Algorithme 6.2 Afficher l'adresse du vecteur

---



```
//Initialisation et affichage de l'adrese du vecteur "v"
int[] v = { 1, 2, 3, 5, 8, 13 };
System.out.println("Vecteur V: " + v);
//w pointera vers le même vecteur que "v", la modification de v[i] modifiera w[i] et
//→ inversement car ce sont les mêmes vecteurs!
int[] w=v;
System.out.println("Vecteur W: " + w);

Vecteur V: [I@2c7b84de
Vecteur W: [I@2c7b84de
```

---

### 6.1.3 Obtenir la taille d'un vecteur

Pour obtenir la taille d'un vecteur, il suffit d'utiliser l'attribut « `length` » du vecteur, cet attribut est non modifiable et est défini lors de la création du vecteur, Il désigne le nombre d'éléments dans le vecteur, c'est à dire l'indice maximum plus un:

---

#### Algorithme 6.3 Taille d'un vecteur

---



```
int[] v = { 1, 2, 3, 5, 8, 13 };
//Affichera la taille du vecteur
System.out.println("Le vecteur v possède " + v.length+" éléments");

Le vecteur v possède 6 éléments
```

---

### 6.1.4 Parcourir un vecteur

Pour parcourir chaque case d'un vecteur, vous pouvez utiliser 2 techniques en Java:

#### Parcourir un vecteur à partir des indices

L'idée consiste à utiliser une boucle « `for` » pour faire varier une variable de 0 jusqu'à l'indice maximum et ainsi accéder à chaque élément du vecteur:

---

**Algorithme 6.4** Parcours d'un vecteur via les indices

---



```
int[] v = { 1, 2, 3, 5, 8, 13 };
//Affichera chaque élément du vecteur
for (int i=0 ; i<v.length ; i++)
    System.out.print(v[i]+ " ");
```

**1 2 3 5 8 13**

---

**Parcourir un vecteur sans utiliser une variable d'indice**

Il existe en Java un boucle « pour » qui consiste à parcourir tous les éléments d'une structure, celle-ci permettra également de parcourir les collections que nous étudierons plus loin.

Voici à quoi ressemble sa syntaxe:

*for (typeElément elem : unVecteur)*

L'instruction « for » exécutera la boucle autant de fois qu'il y a d'éléments dans le vecteur et la variable « elem » contiendra le premier élément du vecteur au premier passage, le deuxième au deuxième passage et ainsi de suite jusqu'au dernier élément du vecteur.

Dans le cadre des collections, que nous étudierons plus loin, nous verrons que cette boucle fonctionne également et qu'il existe aussi une méthode « **foreach** » qui permettra de parcourir tous les éléments d'une liste, d'un ensemble, d'un dictionnaire ou d'un *stream*.

---

**Algorithme 6.5** Parcourir chaque élément d'un vecteur

---



```
int[] v = { 1, 2, 3, 5, 8, 13 };
//elem désigne ici un élément du vecteur et non pas un indice
for (int elem : v)
    System.out.print(elem+ " ");
```

**1 2 3 5 8 13**

---

En utilisant cette boucle, on n'a pas accès aux indices à moins de rajouter une variable que l'on incrémente à chaque passage. Donc en pratique, lorsqu'on a besoin des indices pour accéder aux éléments ou que l'on ne désire pas parcourir tous les éléments du vecteur, il faudra opter pour une boucle « pour » traditionnelle.

**6.1.5 Exemples****Fonction qui retourne le plus grand élément d'un vecteur**

Dans cet exemple, il nous faut une variable pour mémoriser le maximum « maxi », celle-ci sera initialisée au premier élément du vecteur « *v[0]* », ensuite on ajustera cette variable dès qu'un autre élément du vecteur sera plus grand que « *maxi* ». Comme on va parcourir le vecteur qu'à partir de son *2<sup>ème</sup>* élément, on va opter pour la boucle « *for* » classique.

---

**Algorithme 6.6** PseudoCode pour la recherche du plus grand élément d'un vecteur

---

**Require:**  $v.length > 0$

- 1: **function** MAXELEM( $v$ )
- 2:      $maxi \leftarrow v_0$
- 3:     **for**  $i \leftarrow 1, v.length - 1$  **do**
- 4:         **if**  $maxi < v_i$  **then**
- 5:              $maxi \leftarrow v_i$
- 6:         **end if**
- 7:     **end for**
- 8:     **return**  $maxi$
- 9: **end function**

---



---

**Algorithme 6.7** Plus grand élément d'un vecteur en Java

---



```
public static int maxElem(int[] v) {
    assert v.length > 0 : "Le vecteur doit avoir minimim 1 élément";
    int maxi = v[0];
    for (int i = 1; i < v.length; i++)
        if (maxi < v[i])
            maxi = v[i];
    return maxi;
}
```

---

**Fonction qui retourne le nombre de fois qu'une valeur existe dans un vecteur**

Dans cet exemple, il faudra parcourir tous les éléments du vecteur ce qui nous permettra d'utiliser la boucle « for each ». Dès qu'un élément du vecteur sera égal à la valeur recherchée, on incrémentera un compteur:

---

**Algorithme 6.8** PseudoCode: compte le nombre de fois qu'une valeur existe dans un vecteur

---

- 1: **function** GETNBELEM( $v, valeur$ )
- 2:      $cpt \leftarrow 0$
- 3:     **for**  $elem \in v$  **do**
- 4:         **if**  $elem = valeur$  **then**
- 5:              $cpt \leftarrow cpt + 1$
- 6:         **end if**
- 7:     **end for**
- 8:     **return**  $cpt$
- 9: **end function**

---



---

**Algorithme 6.9** Compte le nombre de fois qu'une valeur existe dans un vecteur en Java

---



```
public static int getNbElem(int[] v, int valeur) {
    int cpt = 0;
    for (int elem : v)
        if (elem == valeur)
            cpt++;
    return cpt;
}
```

---

## 6.2 Les structures à plusieurs dimensions

Nous allons nous intéresser plus particulièrement à Java, mais vous trouverez la façon de créer des tableaux à plusieurs dimensions en Python en fin de partie consacrée aux listes(7.1.9).

Un tableau est une structure à 2 dimensions, on doit la voir comme un vecteur de vecteurs. Une structure à 3 dimensions est un vecteur de vecteurs de vecteurs, etc. Généralement dans nos exemples chaque vecteur de la dimension 2 aura la même taille mais ce n'est pas une obligation. On va considérer que la première dimension représentera les lignes de la matrice et la deuxième dimension ses colonnes, de telle sorte que la visualisation suivante correspond à un vecteur de 3 cases contenant chacune un vecteur de 4 cases. J'aurai aussi l'habitude d'utiliser l'indice « *i* » pour parcourir les lignes et l'indice « *j* » pour parcourir les colonnes mais cela dépendra du problème à traiter:

<i>i</i>	<i>j</i>	0	1	2	3
0	0,0	0,1	0,2	0,3	
1	1,0	1,1	1,2	1,3	
2	2,0	2,1	2,2	2,3	

TABLEAU 6.1 – Tableau à 2 dimensions (3\*4)

La visualisation ci-dessous représente une matrice de 3 lignes et 4 colonnes mais en mémoire elle est représentée comme un vecteur qui contient 3 adresses vers un vecteur de 4 cases:

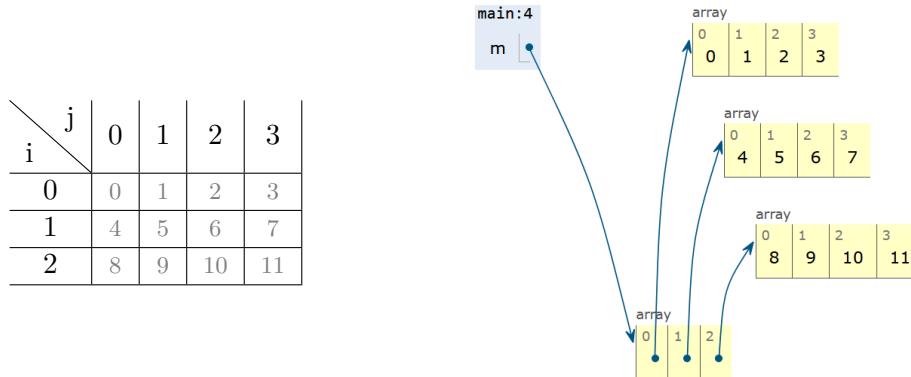


IMAGE 6.1 – Représentation d'un tableau 3\*4 en mémoire

### 6.2.1 Déclarer un tableau d'entier:

Lors de la déclaration de la variable, on ne précise pas la taille du vecteur mais uniquement le type de ses éléments:

```
int [] [] m;
```

La déclaration de cette variable montre bien le vecteur de vecteurs « `int[] []` », la première paire de crochets désigne la 1<sup>ère</sup> dimension (les lignes dans notre représentation) et la deuxième paire de crochets représente la 2<sup>ème</sup> dimension (les colonnes dans notre représentation).

### 6.2.2 Création d'un tableau

Maintenant que l'on a défini une variable de type tableau, on va pouvoir lui assigner ses vecteurs. Pour ce faire, il faudra demander de la mémoire dynamiquement et fournir l'adresse de cette zone dans la variable. Dans l'exemple ci dessous, je crée une matrice de 3 sur 4 d'entiers d'une seule traite:

```
//Création d'une matrice d'entiers de 3 sur 4
m = new int[3][4];
```

Lorsqu'un tableau est créé de cette façon, ses cellules seront initialisées par une valeur par défaut:

- ☛ types numériques ==> « **0** » pour les entiers et « **0.0** » pour les réels
- ☛ type booléen ==> « **false** »
- ☛ pour les classes ==> le pointeur « **null** »

Il est également possible de créer la matrice en commençant par le premier vecteur suivi des autres, ce qui permet de créer une structure avec un nombre de colonnes différent d'une ligne à l'autre:

```
//Création d'un tableau de taille triangulaire:
int[][] m3 = new int[3][];
for (int i=0;i<3;i++)
    m3[i]= new int[i+1];
```

m3	0	1	2
0	0		
1	0	0	
2	0	0	0

### Création d'une matrice initialisée:

Il est possible de créer une variable et de l'initialiser directement lors de sa déclaration. Les valeurs du tableau devront être précisées entre accolades, ligne par ligne (selon notre vision):

```
int[][] m1 = {{0, 1, 2, 3}, {4, 5, 6, 7} , {8, 9, 10, 11}};
```

m1	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

Il est également possible de définir des vecteurs ayant des tailles différentes:

```
int[][] m2 = {{5, 8}, {2, 3, 5}, {1, 1, 2, 3}};
```

m2	0	1	2	3
0	5	8		
1	2	3	5	
2	1	1	2	3

### 6.2.3 Obtenir la taille d'un des vecteurs de la matrice

Comme une matrice est un vecteur de vecteurs, on peut obtenir la taille de chaque vecteur en utilisant son attribut « **length** »:

---

**Algorithme 6.10** Taille des éléments d'une matrice

---



```
int [][] m = {{1, 5, 8, 2}, {2, 3, 5, 8}, {1, 7, 2, 3}};
//Affichera la taille du premier vecteur (nombre de lignes)
System.out.println("La matrice carrée possède "+ m.length + " lignes");
//Affichera la taille du premier élément de vecteur (nombre de colonnes de la 1ère
→ ligne)
System.out.println("La matrice carrée possède "+ m[0].length +" colonnes");

La matrice carrée possède 3 lignes
La matrice carrée possède 4 colonnes
```

---

**6.2.4 Parcourir une matrice**

Pour parcourir chaque case d'une matrice, on peut utiliser la boucle « for » classique ou la boucle « for each ». Il faudra utiliser 2 boucles imbriquées, une boucle principale pour parcourir une dimension et une deuxième boucle pour parcourir l'autre dimension. Vu l'organisation, il est plus intéressant de parcourir la matrice ligne par ligne.

**Parcourir une matrice via ses indices**

L'idée consiste à utiliser 2 boucle « for » imbriquées, la boucle « for » extérieure pour faire varier l'indice de ligne « i » (celui du premier vecteur) et une 2<sup>ème</sup> boucle « for » pour faire varier l'indice de colonne « j », celui du vecteur associé à la *ligne<sub>i</sub>* :

---

**Algorithme 6.11** Parcours d'une matrice via ses indices

---



```
int [][] m = {{1, 5, 8, 2}, {2, 3, 5, 8}, {1, 7, 2, 3}};
for (int i= 0; i<m.length; i++){
    for (int j= 0; j<m[i].length; j++){
        System.out.printf("%3d ",m[i][j]);
    }
    System.out.println();
}
```

1	5	8	2
2	3	5	8
1	7	2	3

---

**Parcourir une matrice via des boucles « for each »**

Si on utilise une double boucle imbriquée « for each », la boucle extérieure renverra des vecteurs et la boucle intérieure les éléments de la matrice:

---

**Algorithme 6.12** Parcourir chaque élément de la matrice

---



```
int[][] m = { { 1, 5, 8, 2 }, { 2, 3, 5, 8 }, { 1, 7, 2, 3 } };
for (int[] v : m) {
    for (int elem : v) {
        System.out.printf("%3d ", elem);
    }
    System.out.println();
}
```

1	5	8	2
2	3	5	8
1	7	2	3

En utilisant cette boucle, on n'a pas accès aux indices à moins de rajouter une variable que l'on incrémente à chaque passage. Donc en pratique, lorsqu'on a besoin des indices pour accéder aux éléments ou que l'on ne désire pas parcourir tous les éléments du vecteur, il faudra opter pour une boucle « pour » traditionnelle.

**6.2.5 Exemples****Procédure qui affiche une matrice dans la console**

Dans ce exemple, on a besoin de parcourir tous les éléments de la matrice ligne par ligne, une structure « for each » s'y prête donc bien:

---

**Algorithme 6.13** Afficher une matrice ligne par ligne en Java

---



```
public static void afficheMat(int[][] m) {
    for (int[] v : m) {
        for (int elem : v)
            System.out.printf("%3d ", elem);
        System.out.println();
    }
}
```

---

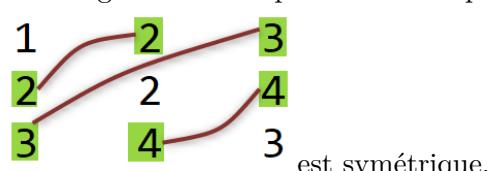
**Fonction qui vérifie qu'une matrice est symétrique**

Cela revient à s'assurer que  $m_{i,j} = m_{j,i}$ , comme on doit s'arrêter dès que la condition n'est plus vérifiée, il faudra utiliser des boucles conditionnelles.

La matrice suivante n'est pas symétrique, on arrête l'algorithme dès qu'on constate que  $m_{2,0} \neq m_{0,2}$ :

1	2	3
2	1	4
4	4	1

Par contre la matrice



---

**Algorithme 6.14** PseudoCode qui vérifie si une matrice carrée est symétrique

---

**Ensure:** m est une matrice carrée d'entiers

```

1: function ESTSYMETRIQUE(m)
2:   sym  $\leftarrow$  true
3:   i  $\leftarrow$  1
4:   while (i  $<$  m.length) and sym do
5:     j  $\leftarrow$  0
6:     while (j  $<$  i) and sym do
7:       sym  $\leftarrow$  mi,j = mj,i
8:       j  $\leftarrow$  j + 1
9:     end while
10:    i  $\leftarrow$  i + 1
11:  end while
12:  return sym
13: end function

```

---

**Algorithme 6.15** Vérifie si une matrice carrée est symétrique (en Java)

---



```

public static boolean estSymetrique(int[][] m) {
    boolean sym = true;
    int i = 1;
    int j;
    while (i < m.length && sym) {
        j = 0;
        while (j < i && sym) {
            sym = m[i][j] == m[j][i];
            j++;
        }
        i++;
    }
    return sym;
}

```

---

## 6.3 Gestion des débordements dans une structure à 1 ou plusieurs dimensions

Un problème récurrent quand on manipule des structures à une ou plusieurs dimensions, c'est de ne pas sortir des limites pour éviter un plantage du programme. Prenons en exemple un vecteur de 5 lignes, lorsqu'on recherche la présence d'un élément dans le vecteur, il faut s'assurer que la variable d'indice reste entre 0 et 4! Dans une recherche de ce type, il faut parcourir le vecteur jusqu'à ce qu'on retrouve l'élément ou qu'on arrive à la fin du vecteur.

```

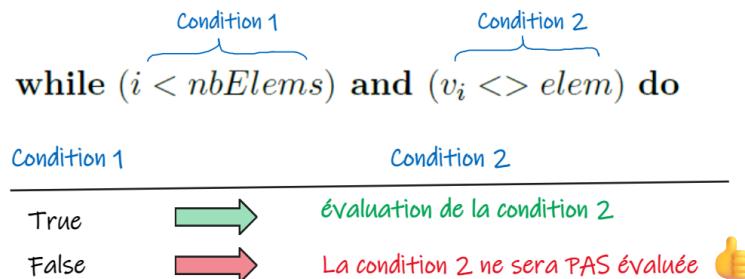
1: function RECHERCHE(v,elem)
2:   i  $\leftarrow$  0
3:   nbElems  $\leftarrow$  v.length
4:   while (i  $<$  nbElems) and (vi  $\neq$  elem) do
5:     i  $\leftarrow$  i + 1
6:   end while
7:   return (i  $\geq$  nbElems)
8: end function

```

---

Intéressons nous de plus près à la condition de la boucle *while* (ligne4), le test comprend un opérateur *AND*, pour que le résultat soit *vrai*, les 2 conditions doivent être à vrai!

Dans la majorité des langages de programmation, lorsque le résultat du test est évident, le reste de l'expression ne sera pas évalué, de ce fait l'ordre des conditions à beaucoup d'importance, voyons cela sur l'exemple précédent:



La condition 2 ne peut pas être évaluée lorsque la condition 1 donne faux, sinon on aurait une exception signalant un débordement car  $v_{nbElems}$  n'existe pas! C'est pour cette raison que les conditions ne peuvent en aucun cas être inversées!

GROSSE ERREUR!!!  
Lorsque ( $i = nbElems$ ) la première condition va générer une exception pour débordement

**while** ( $v_i \neq elem$ ) **and** ( $i < nbElems$ ) **do**

### Langage et code de programmation

Actuellement la majorité des langages de programmation arrête l'évaluation d'une expression lorsque son résultat est connu, mais ce n'est pas le cas dans tous les langages, par exemple en *Delphi* il est possible de désactiver ce comportement pour forcer une évaluation complète!

Cependant rien n'empêche de réécrire le code de telle façon que ce problème n'arrivera plus, quelque soit le langage utilisé:

```

1: function RECHERCHE( $v, elem$ )
2:    $i \leftarrow 0$ 
3:    $nbElems \leftarrow v.length$ 
4:    $trouve \leftarrow false$ 
5:   while ( $i < nbElems$ ) and ! $trouve$  do
6:      $trouve \leftarrow (v_i = elem)$ 
7:      $i \leftarrow i + 1$ 
8:   end while
9:   return  $trouve$ 
10: end function

```

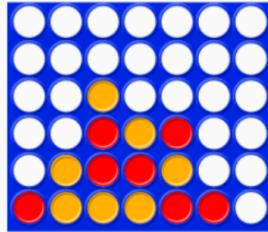
### 6.3.1 Jouer sur la codification pour faciliter les tests de débordement

Dans les jeux qui utilisent un plateau pour positionner des pions comme un damier, un échiquier, un othellier,... on utilise un code pour désigner une case vide, un pion d'un joueur, un pion d'un autre joueur, etc.

Exemple d'un jeu comme puissance 4,

Matrice de 6 lignes sur 7 colonnes

pion jaune => 1  
 pion rouge => 2  
 cas vide => 0



$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 2 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 & 0 & 0 \\ 2 & 1 & 1 & 1 & 2 & 2 & 0 \end{pmatrix}$$

Imaginons une boucle qui vérifie à partir d'une position initiale et dans une direction donnée, si on a 4 pions consécutifs de la même couleur (à partir de la position initiale).

Attention, ce code n'est pas correct pour le jeu de puissance 4 car il ne tient compte que d'une direction à la fois. Par exemple, il ne détectera pas 4 pions de la même couleur si 2 pions sont à gauche de la case initiale et 1 à droite! Rien ne vous empêche d'adapter le code pour qu'il puisse fonctionner dans les tous les cas.

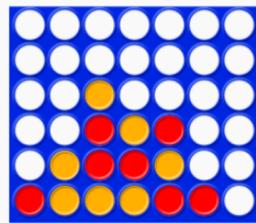
```

1: function SUITEDE4(jeu,posX,posY,couleur,dirX,dirY)
2:   cpt ← 1 // nb pions de la couleur
3:   i ← posX + dirX // indice en x
4:   j ← posY + dirY // indice en y
5:   while (i ≥ 0) and (i < 6) and (j ≥ 0) and (j < 7) and jeui,j = couleur and cpt < 4 do
6:     i ← i + dirX
7:     j ← j + dirY
8:     cpt ← cpt + 1
9:   end while
10:  return cpt = 4 // retourne true si 4 à la suite
11: end function
```

On constate la lourdeur de la condition qui doit vérifier les débordements à gauche, à droite, en haut et en bas car la direction peut être modifiée. Pour éviter ce long test, une idée consiste à rajouter des sentinelles (valeurs de fin) ce qui augmentera la taille de la matrice mais allégera fortement le test étant donné qu'un débordement ne sera plus possible car la boucle s'arrêtera sur les valeurs sentinelles (-1):

Matrice de 8 lignes sur 9 colonnes

pion jaune => 1  
 pion rouge => 2  
 cas vide => 0  
**bordure** => -1



Ajout d'une bordure comme sentinelle d'arrêt

0	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	0	0	0	0	0	0	0	-1
2	-1	0	0	0	0	0	0	0	-1
3	-1	0	0	1	0	0	0	0	-1
4	-1	0	0	2	1	2	0	0	-1
5	-1	0	1	2	2	1	0	0	-1
6	-1	2	1	1	1	2	2	0	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8

```

1: function SUITEDE4(jeu, posX, posY, couleur, dirX, dirY)
2:   cpt <- 1 // nb pions de la couleur
3:   i <- posX + dirX // indice en x
4:   j <- posY + dirY // indice en y
5:   while jeu[i,j] = couleur and cpt < 4 do
6:     i <- i + dirX
7:     j <- j + dirY
8:     cpt <- cpt + 1
9:   end while
10:  return cpt = 4 // retourne true si 4 à la suite
11: end function

```

Le résultat est clairement visible, on ne doit plus se préoccuper des débordements éventuels!

### 💡 Truc

Il est parfois possible de définir des valeurs sentinelles aux extrémités d'une structure à 1 ou plusieurs dimensions pour ne pas devoir tester les débordements des variables d'indices et ainsi alléger la condition d'une boucle de parcours.



## Les listes, tuples et dictionnaires en Python

En Python tout est objet, on ne va pas retrouver des structures de tableaux comme en C ou en Java, par contre Python possède plusieurs classes intégrées au langage qui vont permettre de mémoriser des structures de liste, de tuple, d'ensemble et de dictionnaire. Chacune de celles-ci vont avoir un ensemble de méthodes et d'opérateurs qui vont vous faciliter la tâche. En voici un bref aperçu avant de les voir plus en détail:

« **list** »: permet de mémoriser une liste modifiable d'éléments. Elle possède plusieurs méthodes qui facilitent fortement sa manipulation par rapport à un vecteur classique vu au chapitre précédent.

```
#La création d'une liste nécessite juste la présence de 2 crochets
l = [2,3,5,7,11]
#On peut rajouter des éléments
l.append(13)
```

« **tuple** »: permet de mémoriser une liste non modifiable d'éléments.

```
#On peut définir un tuple avec ou sans les parenthèses
t1 = 2, 3, 5
t2 = (2, 3, 5)
print(t1 == t2) #affichera true car t1 est identique à t2
```

« **set** »: représente un ensemble non ordonné d'objets sans doublon:

```
#Exemple d'ensemble
ens1 = set([2, 3, 5])
ens2 = set([3, 4, 6])
#affiche l'union des 2 ensembles
print(ens1 | ens2) #affichera [2, 3, 4, 5, 6]
```

« **dict** »: c'est une structure clé-valeur, à partir d'un objet identifiant (la clé), on associe un objet cible (la valeur).

```
#un dictionnaire pour traduire du français vers l'anglais
fr_en={'cours':'course','stage':'internship','professeur':'teacher'}
print(fr_en['cours'])# affichera 'course'
```

Regardons maintenant plus précisément chacune de ces classes intégrées au langage Python en commençant par les listes suivi des tuples, des ensembles et des dictionnaires.

## 7.1 Les listes en Python

Une liste en Python peut contenir en même temps plusieurs éléments de type différent, elle est aussi modifiable (mutable) et on peut rajouter ou supprimer des éléments sans problème. Voici quelques exemples de listes:

```
l1 = [] #liste vide
l2 = [2, 3, 5] #liste de 3 entiers
#la liste l3 contient un string, un tuple et une autre liste
l3 = ['Hello', (2, 3), ['W', 'X', 'Y', 'Z'] ]
print( len(l3) ) #Affichera la taille de l3 soit 3
print( len(l3[2]) ) #Affichera 4, c&gt;ad la taille de la liste ['W', 'X', 'Y', 'Z']
```

### 7.1.1 Accéder aux éléments de la liste

Les éléments d'une liste sont accessibles via un indice qui peut s'exprimer de 0 à  $taille - 1$  pour aller du premier élément jusqu'au dernier élément. Il est également possible d'utiliser des indices négatifs de  $-1 \rightarrow -taille$  pour accéder du dernier élément jusqu'au premier élément:

v	1	2	3	5	8	13
indices positifs:	0	1	2	3	4	5
indices négatifs:	-6	-5	-4	-3	-2	-1

TABLEAU 7.1 – Indices d'une liste en Python

Ainsi `v[2]` et `v[-4]` désignent la même cellule au sein de la liste.

Pour avoir la taille d'une liste, il faut utiliser la fonction intégrée: `len( maListe )` ainsi, `len( v )` donnera 6

### 7.1.2 Copier une liste

La copie d'une liste consiste à créer une nouvelle liste qui contiendra les mêmes éléments que la première. Cependant comme une liste contiendra des référents vers des objets (rappelez-vous en Python tout est objet), ce sont ces référents qui seront copiés dans la nouvelle liste.

Ainsi une copie classique, dite peu profonde « `shallow copy` », revient à copier la liste avec ses référents mais sans dupliquer ses éléments. On parle sinon de copie profonde « `deep copy` », lorsque la liste est clonée ainsi que tous ses éléments.

#### Copie classique « peu profonde » d'une liste:

Pour cloner une liste, il suffit de faire `maListe[:]`, cependant il ne s'agit pas d'un clonage profond, seule la liste est dupliquée, pas ses éléments! Rappelez-vous qu'en Python tout est objet ainsi une liste contiendra des référents (pointeurs) vers ses objets, le clonage d'une liste copiera les référents mais ne clonera pas ses objets. Ce qui ne posera pas de problème pour des objets immuables mais il faudra être attentif lorsqu'une liste possèdera des éléments mutables!

```
v1 = [1, 2, 3, 5, 8, 13] #liste d'éléments immuables
```

```
v2 = v1[:] #v2 sera une nouvelle liste identique à v1
```

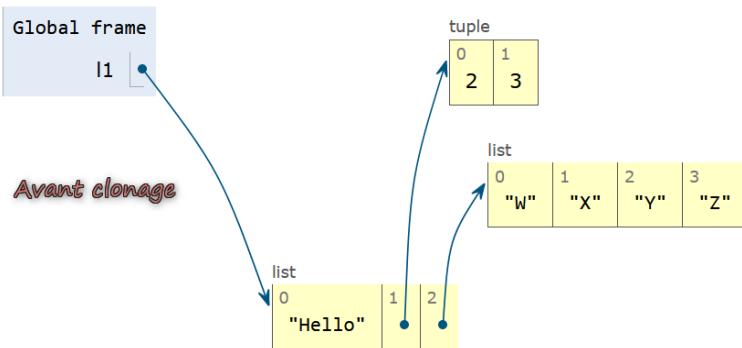
```
l1 = ['Hello', (2, 3), ['W', 'X', 'Y', 'Z'] ]
```

```
l2 = l1[:] #l2 sera aussi une copie de l1 mais attention l1[2] et l2[2]
```

→ pointent sur le même élément mutable ainsi la modification de la liste

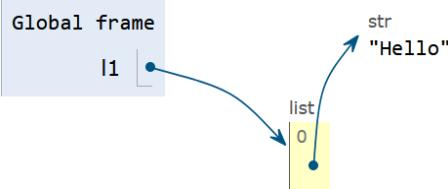
→ l1[2] modifiera aussi l2[2]

Regardons plus précisément la liste « l1 » et ensuite sa copie l2. Commençons par regarder la représentation graphique de la liste « l1 » avant son clonage:

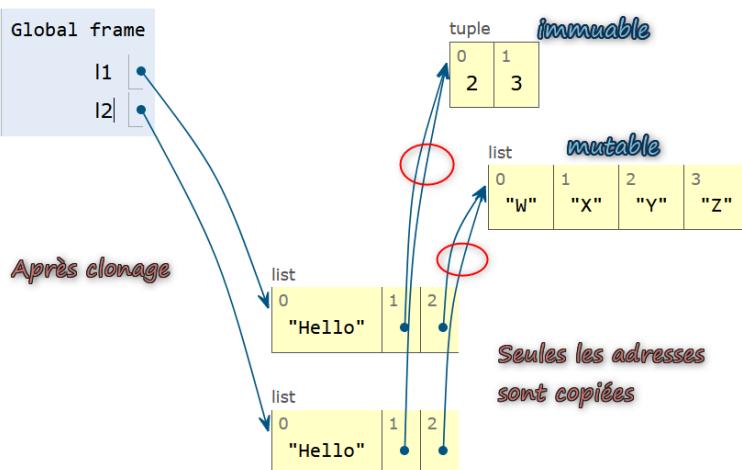


### Remarque

Les chaînes de caractères et les entiers sont aussi des objets référencés mais pour alléger la visualisation graphique ces objets immuables ne sont pas représentés correctement, par exemple la chaîne « Hello » devrait être visualisée de la façon suivante:

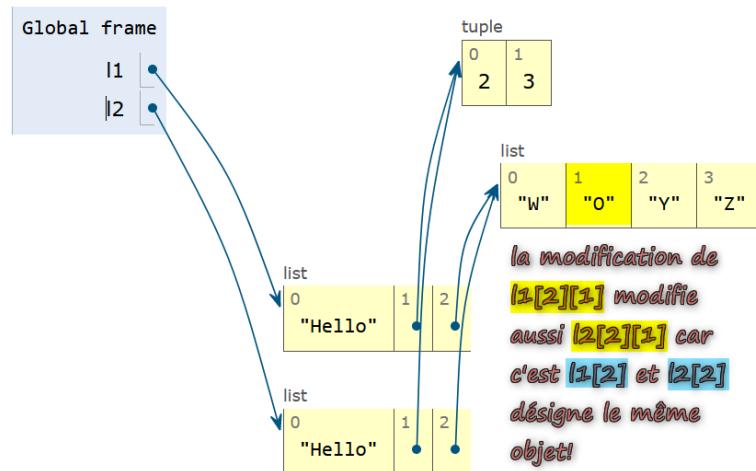


Après le clonage de la liste « l1 », on peut constater que l2 pointe bien sur une nouvelle liste mais ses éléments pointent vers les mêmes objets (idem pour la chaîne « Hello » même si le graphique ne le montre pas):



Le tuple n'est pas modifiable mais la liste « l2[2] » est mutable ainsi si je modifie un de ses éléments en faisant:

`l1[2][1]='O'`, on aurait la modification suivante:



On constate que c'est la même liste qui est modifiée. Pour les éléments immuables le problème ne se pose pas, par exemple il n'est pas possible de modifier un tuple et pour un entier si on le modifie, un nouvel entier sera créé. Regardons encore ceci sur un exemple où tous les objets sont correctement visualisés:

---

#### Algorithme 7.1 Modification d'un élément immuable d'une liste

---

```

1 l1 = [4,9,7]
2 l2 = l1[:] #Copie de l1
3 l2[0] +=2 #Modification du 4 en 6 (seule la liste l2 sera modifiée)

```

---

Voici la représentation avant et après l'exécution de la ligne 3:

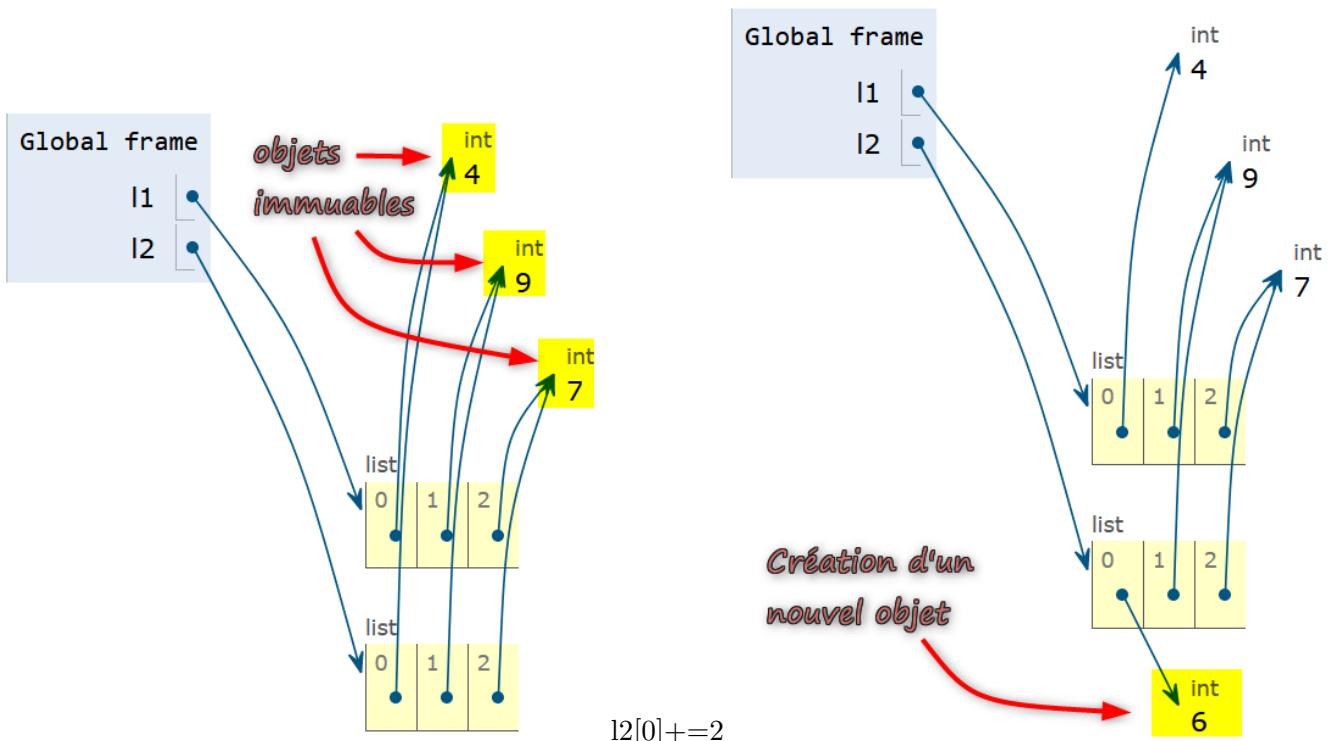


IMAGE 7.1 – Modification d'un objet immuable d'une cellule d'une liste

On peut clairement visualiser sur le graphique que la liste `l2` est maintenant différente de `l1` car la

modification de `l2[0]` va remplacer le référent par celui du nouvel entier (6).

### Copie profonde d'une liste « deep copy »

Comme on l'a vu précédemment, une copie profonde d'une liste pas nécessaire lorsque ses éléments sont immuables par contre si ses éléments sont mutables, cela peut être utile. Cependant la copie profonde n'est pas une mince affaire car cela reviendrait à faire une copie profonde de chacun des ses éléments et donc les éléments de ses derniers et ainsi de suite. Heureusement qu'il existe un module « `copy` » avec une fonction « `deepcopy` » que fait le job pour vous. Maintenant il est assez peu fréquent de devoir faire une copie profonde d'une liste.

---

#### Algorithme 7.2 Copie profonde d'une liste

---

```
1 import copy
2 l1 = [[2,3],[4,5]] #L1 est une liste de listes
3 l2 = copy.deepcopy(l1) #Copie profonde de l1
```

---

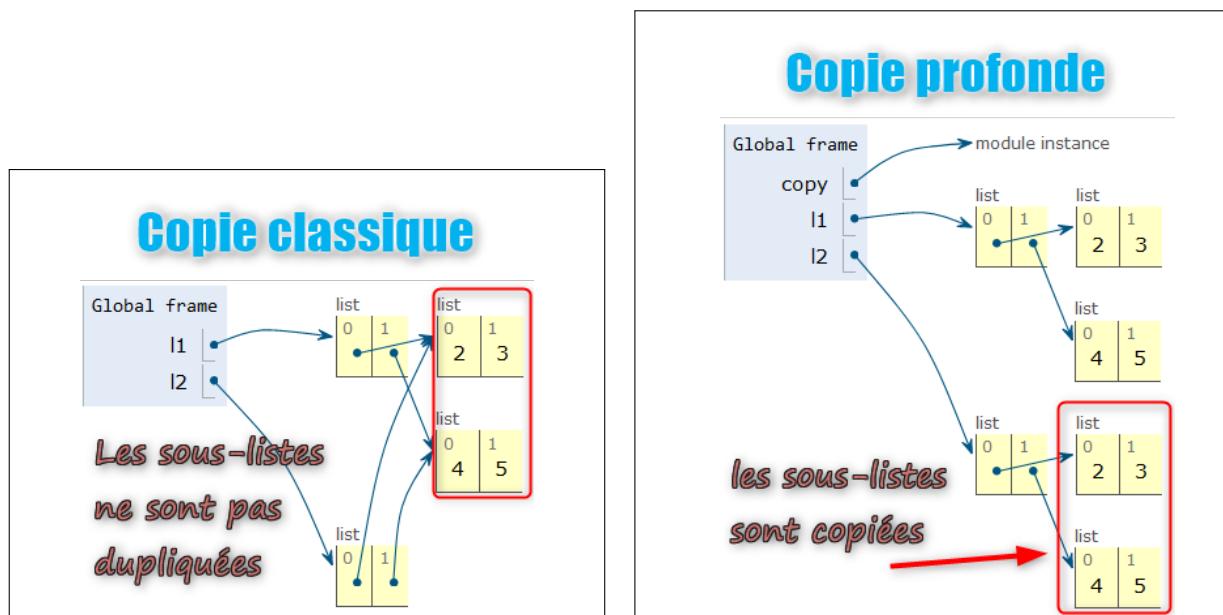


IMAGE 7.2 – Copie classique / copie profonde

### 7.1.3 Copier une portion d'une liste (slice)

Nous venons de voir comment copier une liste entière à partir de l'opérateur de « slice » `[:]`, mais il ne s'agit qu'un cas particulier de cet opérateur, voici sa syntaxe complète:

`maListe[indicedébut : indicefin]`

L'opérateur permet de copier une liste de l'indice de début compris jusqu'à l'indice de fin non compris. Voici quelques remarques à bien comprendre:

- ☛ si l'indice de début n'est pas précisé, par défaut il sera égal à 0
- ☛ si l'indice de fin n'est pas précisé, par défaut il sera égal à la taille de la liste
- ☛ un indice peut être exprimé par sa forme positive ou négative

- si l'indice de début désigne une case supérieure ou égale à l'indice de fin, une liste vide sera renvoyée

Exemples:

---

**Algorithme 7.3** Copie partielle d'une liste
 

---

v	1	2	3	5	8	13
indices positifs:	0	1	2	3	4	5
indices négatifs:	-6	-5	-4	-3	-2	-1

```
v = [1, 2, 3, 5, 8, 13]
l1 = v[2: 4] # l1 recevra [3, 5]
l2 = v[1:-2] # l2 recevra [2, 3, 5]
l3 = v[4: 2] # l3 recevra []
l4 = v[3: ] # l4 recevra [5, 8, 13]
l5 = v[ : -4] # l5 recevra [1, 2]
```

---

## 7.1.4 Modification d'une liste

### Modification d'une liste à partir d'un indice

Vous pouvez modifier facilement un élément de la liste à partir de son indice mais celui-ci doit rester dans les bornes de 0 à taille-1.

```
v = [1, 4, 3, 8, 5, 20]
#Les résultats indiqués montre l'exécution à partir de la liste v
→ initiale!
v[2] = 30 # v = [1, 4, 30, 8, 5, 20]
v[6] = 60 # ERREUR car l'indice 6 est en dehors de la liste
v[0] =[10, 11] # insère une liste en position 0: [[10, 11], 4, 3,
→ 8, 5, 20]
```

### Modification d'une liste avec l'opérateur de « slice » [:]

On peut également utiliser l'opérateur de « slice »[:] dans le membre de gauche d'une assignation pour indiquer l'endroit où insérer ou modifier une liste d'éléments:

$$\text{maListe}[indice_{\text{début}} : indice_{\text{fin}}] = \text{un objet itérable}$$

Le membre de droite doit être une objet *itérable*, c'est un objet qui peut renvoyer chacun de ses membres les uns à la suite des autres, comme une liste, un tuple, un dictionnaire, range(...),...

Il est ainsi possible de rajouter, de supprimer, de modifier une liste existante:

v	10	4	-3	9
indices positifs:	0	1	2	3
indices négatifs:	-4	-3	-2	-1

Chaque exemple du tableau ci-dessous, s'exécute à partir de la liste « v » initiale

Opérations	Code Python	Résultat
remplacer un élément	v[2:3] = [50]	[10, 4, 50, 9]
insérer une liste en position 2	v[2: 3] = [50, 60]	[10, 4, 50, 60, 9]
supprimer un élément en position 1	v[1:2] = []	[10, -3, 9]
remplacer la portion [4, -3] par [50, 60]	v[1:3] = [50, 60]	[10, 50, 60, 9]
insérer en fin de liste	v[len(v):] = [10, 11]	[10, 4, -3, 9, 10, 11]
insérer en début de liste	v[:0] = [2, 3]	[2, 3, 10, 4, -3, 9]

TABLEAU 7.2 – Opérations sur une liste avec l'opérateur de slice

### 7.1.5 Modification d'une liste à partir de méthodes ou fonctions intégrées

La classe « list » possède plusieurs méthodes ou fonctions intégrées pour rajouter ou supprimer des éléments, ainsi que trier ou inverser une liste. L'opérateur de slice vu précédemment permet de réaliser la plupart de ses opérations mais en utilisant une syntaxe pas toujours très compréhensible, c'est pourquoi il est parfois préférable d'utiliser les méthodes comme « append », « insert » ou la fonction intégrée « del » pour rajouter ou supprimer des éléments dans une liste.

v	10	4	-3	9
indices positifs:	0	1	2	3
indices négatifs:	-4	-3	-2	-1

Chaque exemple du tableau ci-dessous, s'exécute à partir de la liste « v »:

Opérations	Code Python	Résultat
intercaler une liste en position 2	v.insert(2, [50, 60])	[10, 4, 50, 60, -3, 9]
supprimer un élément en position 1	del (v[1])	[10, -3, 9]
supprime une portion d'une liste	del (v[1:3])	[10, 9]
supprimer la première occurrence d'un élément	v.remove(4)	[10, -3, 9]
insérer un élément en fin de liste	v.append(10) v.append([10, 20]) v.insert(len(v),10)	[10, 4, -3, 9, 10] [10, 4, -3, 9, 10, 20] [10, 4, -3, 9, 10]
insérer plusieurs éléments en fin de liste	v.extend( [10, 20]) v.extend(range(1,4))	[10, 4, -3, 9, 10, 20] [10, 4, -3, 9, 1, 2 ,3]
inverser une liste	v.reverse()	[9, -3, 4, 10]
trier une liste	v.sort()	[-3, 4, 9, 10]
renvoyer une copie triée	sorted(v)	[-3, 4, 9, 10] (nouvelle liste)

TABLEAU 7.3 – Méthodes et fonctions intégrées pour modifier une liste

### 7.1.6 Autres méthodes sur une liste

Voici quelques autres méthodes ou opérateurs applicable sur une liste:

v	10	4	-3	9	4
indices positifs:	0	1	2	3	4
indices négatifs:	-5	-4	-3	-2	-1

Opérations	Code Python	Résultat
appartenance à la liste	3 <b>in</b> [4, 9, 3, 21]	True
non appartenance à la liste	1 <b>not in</b> [4, 9, 3, 21]	True
concaténation de 2 listes	[4, 9, 8] <b>+</b> [10, 4, 20]	[4, 9, 8, 10, 4, 20] (nouvelle liste)
répéter plusieurs fois une liste	[1, 2] <b>*</b> 4	[1, 2, 1, 2, 1, 2, 1, 2] (nouvelle liste)
élément minimum	<b>min</b> ([4, 9, 3, 21])	3
élément maximum	<b>max</b> ([4, 9, 3, 21])	21
index d'un élément dans une liste	v. <b>index</b> (9) v. <b>index</b> (4)	3 1
nombre de fois qu'un élément existe dans la liste	v. <b>count</b> (4)	2

TABLEAU 7.4 – Autre opérations sur une liste

### 7.1.7 Parcourir les éléments d'une liste

#### Algorithme 7.4 Parcourir les éléments d'une liste

```
v = [1, 4, 3, 8, 5]
#Parcourir tous les éléments
for elem in v:
    print(elem, end=', ') # Affichera: 1, 4, 3, 8, 5,
#On peut utiliser enumerate pour afficher les positions
for i, elem in enumerate(v):
    print(f'{i}: {elem}', end=', ') #Affichera: 0: 1, 1: 4, 2: 3,
    ↪ 3: 8, 4: 5,
#Avec enumerate par défaut l'indice initial sera égal à 0, mais on
↪ peut l'initialiser à une autre valeur, il s'incrémentera de 1
↪ pour chaque élément de la liste
for i, elem in enumerate(v,10):
    print(f'{i}: {elem}', end=', ') #Affichera: 10: 1, 11: 4, 12:
    ↪ 3, 13: 8, 14: 5,
```

### 7.1.8 Résumé

Opérations	Explications	Exemples
[]	Crée une liste vide	v = []
len	retourne la taille d'une liste	len(v)
[indiceDébut : indiceFin] [:]	copie classique d'une portion de liste copie classique d'une liste	w = [2:4] w = v[:]
copy.deepcopy	copie en profondeur d'une liste	w = copy.deepcopy(v)
append	Ajoute un seul élément en fin de liste	v.append(5)
extend	Ajoute un ou plusieurs éléments en fin de liste (iterable)	v.extend([2, 4, 6])
insert	Insère un nouvel élément à une position dans la liste	v.insert(2, 99)
del	supprime un élément d'une liste supprime une portion d'une liste	del(v[2]) del(v[1:4])
remove	recherche une valeur dans une liste et la supprime	v.remove(valeur)
reverse	inverse une liste	v.reverse()
sort	trie une liste de manière croissante	v.sort()
sorted	renvoie une copie triée d'une liste	sorted(v)
+	concaténation de 2 listes	v1+v2
*	duplicte une liste un nombre de fois	v = [1 , 0] *4
min / max	recherche l'élément minimum ou maximum	min(v) et max(v)
index	retourne la première position d'une valeur dans une liste	v.index(3)
count	compte le nombre de fois qu'un élément appartient à une liste	v.count(3)
in et not in	indique si un élément appartient (ou non) à la liste	3 in v et 3 not in v

TABLEAU 7.5 – Résumé des opérations sur une liste

### 7.1.9 Tableaux à plusieurs dimensions

Un tableau à 2 dimensions est en réalité une liste contenant des listes, la dimension de la 1<sup>ère</sup> liste peut être vue comme étant le nombre de lignes d'une matrice et la dimension des autres listes ( si de même taille) comme étant son nombre de colonnes.

#### Création d'un tableau initialisé

```
#Création d'une matrice initialisée de 3 lignes et 4 colonnes
m = [[0, 1, 2, 3], [4, 5, 6, 7], [2, 7, 1, 9]]
```

Ce qui revient à créer la matrice suivante:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 2 & 7 & 1 & 9 \end{pmatrix}$$

## Création d'un tableau dynamiquement

On peut bien entendu partir d'une liste vide « `m = []` » et de lui rajouter dynamiquement des éléments en faisant des ajouts « `m.append([...])` ». Cependant si on connaît la taille de la matrice, on peut la créer et l'initialisée par une « *list comprehension* » et ensuite lui fournir des valeurs:

```

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ #Création d'une matrice initialisée avec des } 0$$

nbLignes, nbColonnes = 3 , 4
m = [[0 for j in range(nbColonnes)] for i in range(nbLignes)]
```

ou s'il est possible de l'initialiser directement selon une construction comme pour la matrice suivante:

```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix} \text{ #Création d'une matrice initialisée avec des } 0$$

nbLignes, nbColonnes = 3 , 4
m = [[nbColonnes * i + j for j in range(nbColonnes)] for i in
      range(nbLignes)]
```

## Accès aux éléments d'un tableau

L'accès aux éléments du tableau se fait par 2 doubles accolades « `m[ligne][colonne]` », prenons un exemple d'une fonction qui retourne la somme de deux matrices dans une troisième:

$$m3 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 2 \end{pmatrix} + \begin{pmatrix} 8 & 2 & 1 \\ 2 & 4 & 5 \end{pmatrix} \Rightarrow \begin{pmatrix} 9 & 4 & 4 \\ 6 & 9 & 7 \end{pmatrix}$$

On peut réaliser cette tâche avec l'écriture d'une « *list comprehension* »:

---

### Algorithme 7.5 Somme de 2 matrices en Python v1

---

```
def sommeMat(m1,m2):
    '''m1 et m2 sont deux matrices de même taille
       la fonction retourne une nouvelle matrice ou
       m3[i][j] = m1[i][j] + m2[i][j]'''
    return [[m1[i][j] + m2[i][j] for j in range(len(m1[0]))]
           for i in range(len(m1))]
```

---

Maintenant si la matrice  $m1 \leftarrow m1 + m2$ , il ne faut pas créer de troisième matrice, on peut donc utiliser 2 boucles « pour »:

---

### Algorithme 7.6 Somme de 2 matrices en Python v2

---

```
def sommeMat2(m1,m2):
    '''m1 et m2 sont deux matrices de même taille
       la fonction retourne
       m1[i][j] = m1[i][j] + m2[i][j]'''
    for i in range(len(m1)):
        for j in range(len(m1[0])):
            m1[i][j] = m1[i][j] + m2[i][j]
    return m1
```

---

## 7.2 Les tuples

Après avoir vu les listes, ce ne sera pas difficile de comprendre le fonctionnement des tuples car il s'agit d'une liste non modifiable. La plupart des opérations vues précédemment fonctionnent aussi sur les tuples à l'exception des méthodes de modification. Par exemple on peut accéder aux éléments d'un tuple ou connaître sa taille de la même manière qu'avec une liste:

```
t = (2, 3, 5)#création d'un tuple
print(len(t))#affichera 3
print(t[1:]) #affichera (3, 5)
print(t[0]) #affichera 2
```

### 7.2.1 Création d'un tuple

Un tuple peut contenir de 0 à plusieurs éléments, bien entendu vous retrouverez plus souvent des tuples avec au moins de 2 éléments. Une liste s'initialise avec des crochets [], alors qu'un tuple s'initialise avec des parenthèses « () » voire même sans les parenthèses mais s'il ne contient qu'un élément il faudra lui rajouter une virgule comme le montre les exemples suivants:

```
#On peut définir un tuple avec ou sans les parenthèses
t1 = 2, 3, 5
t2 = (2, 3, 5)
t3 = () #Un tuple vide
#Un tuple avec un seul élément
t4 = (5,) #Il faut une virgule pour le distinguer d'un entier
```

### 7.2.2 Empaqueter et dés-empaqueter un tuple

Lorsque vous écrivez « 1 , 2 , 3 » pour empaqueter ces 3 nombres dans un tuple. Mais vous pouvez aussi extraire les éléments d'un tuple pour les mettre dans des variables. Si le nombre de variables est inférieur à la taille du tuple, Python permet d'ajouter « \* » devant un des paramètres pour que celui-ci absorbe les éléments supplémentaires sous forme d'une liste:

```
t1 = 2, 3, 5, 6 #Empaqueter
a, b, c, d = t1 #Dés-empaqueter
print(a,b,c,d) #Affichera 2 3 5 6
#Absorption des éléments en trop dans une liste
a, *b, c = t1
print(a,b,c) #Affichera 2 [3,5] 6
a, b, *c = t1
print(a,b,c) #Affichera 2 3, [5, 6]
```

Grâce à cette notion de tuple, Python permet d'écrire très facilement l'initialisation de plusieurs variables, l'échange entre 2 variables et renvoyer plusieurs éléments en retour d'une fonction « *return v1, v2* »:

```
a, b, c = 1, 2, 3 #initialisation de plusieurs variables
a, b = b, a #Swap entre 2 variables
```

### 7.2.3 Parcourir les éléments d'un tuple

---

#### Algorithme 7.7 Parcourir les éléments d'un tuple

---

```
t1 = (2, 'hello', [2,4])

#Parcourir tous les éléments
for elem in t1:
    print(elem, end=', ') # Affichera: 2, hello, [2, 4],

#On peut utiliser enumerate pour afficher les positions
for i, elem in enumerate(t1):
    print(f'{i}: {elem}', end=', ') #Affichera: 0: 2, 1: hello, 2:
    ↵ [2, 4],
```

---

## 7.3 Les ensembles

Un ensemble en Python représente une collection d'objets sans doublons. ceux-ci doivent être immuable et *hashable*<sup>1</sup> comme les entiers, les réels, les strings, les tuples,...

### 7.3.1 Crédation d'un ensemble

La création d'un ensemble passe par la classe « `set` », il n'y a pas d'opérateur spécifique pour créer un ensemble comme c'est le cas avec les listes et les tuples. Il faut appeler le constructeur « `set` » en lui précisant un élément *itérable* en paramètre, les éventuels doublons seront d'office écartés:

```
ens1 = set([7,9,21,4])#avec une liste en entrée
print(ens1) # Affichera {9, 4, 21, 7}
ens2 = set((11,9,10,4,90))#avec un tuple en entrée
print(ens2) # Affichera {4, 9, 10, 11, 90}
ens3 = set(range(10))#avec un range en entrée
print(ens3) {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

---

1. *hashable*: `hash(objet)` va donner une valeur numérique pour cet objet. Deux objets égaux auront la même valeur de hashing mais l'inverse n'est pas le cas, une même valeur de hashing peut correspondre à deux objets différents. Une bonne fonction de hashing doit avoir une bonne répartition des valeurs pour éviter d'avoir trop d'éléments différents avec la même valeur de hashing.

### 7.3.2 Opérations sur un ensemble

```
ens1 = set([7,9,21,4])
ens2 = set([11,9,10,4])
```

Les exemples ne dépendent pas les uns des autres et s'exécutent donc à partir des données initiales!

Opérations	Code Python	Résultat
appartenance	7 in ens1	True
non appartenance	20 not in ens1	True
taille d'un ensemble	len(ens1)	4
ajout d'un élément	ens1.add(5) ens1.add(9)	{4, 5, 7, 9, 21} {4, 7, 9, 21}
suppression d'un élément	ens1.remove(9)	{4, 7, 21}
éléments minimum et maximum	min(ens1) max(ens1)	4 et 21
union entre 2 ensembles	ens1   ens2	{4, 7, 9, 10, 11, 21}
intersection entre 2 ensembles	ens1 & ens2	{9, 4}
différence entre 2 ensembles	ens1 ^ ens2	{21, 7, 10, 11}

TABLEAU 7.6 – Opérations sur un ensemble

### 7.3.3 Parcourir les éléments d'un ensemble

Les éléments d'un ensemble n'ont pas d'ordre mais on peut les parcourir avec une boucle « *for* »:

---

#### Algorithme 7.8 Parcourir les éléments d'un ensemble

---

```
ens1 = set([7,9,21,4])

#Parcourir tous les éléments
for elem in ens1:
    print(elem, end=' ', ) # Affichera: 9, 4, 21, 7

#On peut utiliser enumerate même si l'ordre n'a pas d'intérêt
for i, elem in enumerate(ens1):
    print(f'{i}: {elem}', end=' ', ) #Affichera: 0: 9, 1: 4, 2: 21, 3: 7,
```

---

## 7.4 Les dictionnaires

Les dictionnaires sont des structures de mémorisation très intéressantes et performantes, il s'agit de paires « clé-valeur ». La clé sera un objet *immutable* et *hashable* qui devra être unique pour pouvoir identifier l'élément associé. La valeur pourra être n'importe quel objet. Les valeurs (objets) d'un dictionnaire n'ont pas d'ordre, les clés ne sont pas ordonnancées non plus. Regardons maintenant comment créer et utiliser un dictionnaire.

### 7.4.1 Création d'un dictionnaire

Pour définir un dictionnaire, il suffit d'utiliser 2 accolades {}, il peut être vide au départ ou on peut l'initialiser avec des paires clé-valeur.

#### Algorithme 7.9 Crédation et initialisation d'un dictionnaire

```
#Dictionnaire vide
professeurs = {}
professeurs['vo'] = ('Didier', 'Van Oudenhove')
#Dictionnaire initialisé
fr_en={'cours':'course','stage':'internship','professeur':'teacher'}
#Ajout d'un mot dans le dictionnaire
fr_en['embarqué'] = 'embedded'
```

### 7.4.2 Opérations sur un dictionnaire:

fr\_en={'cours':'course','stage':'internship','professeur':'teacher'}

Les exemples ne dépendent pas les uns des autres et s'exécutent donc à partir des données initiales!

Opérations	Code Python	Résultat
taille	len(fr_en)	3
ajout d'un élément	fr_en['slice'] = 'portion'	ajout la paire 'slice' → 'portion'
suppression d'un élément	del(fr_en['slice'])	supprime la paire
existence d'une clé	'cours' in fr_en	True
obtenir une valeur à partir d'une clé	fr_en['cours'] fr_en.get('cours', 'Vide') fr_en.get('brol', 'Vide')	'course' 'course' 'Vide'
obtenir la liste des clés	list(fr_en.keys())	['cours', 'stage', 'professeur']
obtenir la liste des valeurs	list(fr_en.values())	['course', 'internship', 'teacher']
obtenir la liste des paires	list(fr_en.items())	[('cours', 'course'), ('stage', 'internship'), ('professeur', 'teacher')]
copie classique d'un dictionnaire	fr_en.copy()	duplicte le dictionnaire mais pas ses éléments
copie profonde d'un dictionnaire	copy.deepcopy(fr_en)	duplicte le dictionnaire et ses éléments
fusion de 2 dictionnaires	fr_en.update(unDict)	« fr_en » recevra les nouvelles paires et les clés existantes seront mise à jour avec les valeurs du dictionnaire « unDict »

TABLEAU 7.7 – Opérations sur un dictionnaire

#### Remarque

Les méthodes « *keys()*, *values()*, *items()* » retournent des objets dynamiques qui changeront lorsque le dictionnaire sera modifié, c'est pour cette raison que dans les exemples ci-dessus, on

| les transforme en une liste en faisant « `list(monDict.keys())` ».

| L'ordre des éléments renvoyés par ces méthodes sera indéterminé.

### 7.4.3 Parcourir un dictionnaire

Comme un dictionnaire possède des clés et des valeurs, vous pouvez exploiter les méthodes `keys`, `values` et `items` avec une boucle « `for` »:

---

#### Algorithme 7.10 Parcourir les éléments d'un dictionnaire

---

```
fr_en={'cours':'course','stage':'internship','professeur':'teacher'}

#Parcourir tous les éléments
for clef, val in fr_en.items():
    print(clef, val, end=', ') # Affichera: cours courses, stage internship,
    → professeur teacher,

#Parcourir les clés version1
for clef in fr_en:
    print(clef , end=', ') # Affichera: cours, stage, professeur,

#Parcourir les clés version2
for clef in fr_en.keys():
    print(clef , end=', ') # Affichera: cours, stage, professeur,

#Parcourir les valeurs
for valeur in fr_en.values():
    print(valeur , end=', ') # Affichera: courses, internship, teacher,

#Vous pouvez aussi utiliser "enumerate" même si les positions renvoyées n'ont pas
→ bcp d'intérêt car les éléments d'un dictionnaire n'ont pas d'ordre
for i, clef in enumerate(fr_en):
    print(i, clef , end=', ') #Affichera: 0 cours, 1 stage, 2 professeur, 3 brol,
```

---



# Chapitre 8

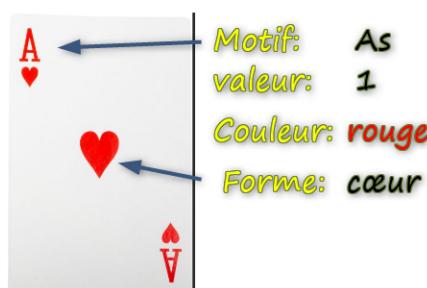
## La notion d'enregistrements ou de structure

L'idée consiste à regrouper au sein d'une même variable plusieurs données, un peu comme un enregistrement dans une base de données. Par exemple, une personne possède un nom, un prénom et une date de naissance, il serait pratique d'avoir une structure pour permettre de mémoriser ces 3 informations au sein d'une même variable.

- ➔ si ces données peuvent être modifiées, la bonne solution consisterait à créer une classe(14)
- ➔ si par contre les données ne sont pas modifiables, on peut utiliser des structures comme des tuples de Python (pas idéal car les champs n'ont pas de nom) ou des *Records* en Java.

Imaginons qu'on désire modéliser un jeu de cartes, chaque carte possède 4 informations:

- ➔ une forme (pique, trèfle, carreau, cœur)
- ➔ une motif (As, Deux,... , Dix, Valet, Dame, Roi)
- ➔ la valeur du motif qui peut dépendre du jeu auquel on joue
- ➔ une couleur qui est associée à la forme (noir, rouge)



Pour modéliser le motif, la forme et la couleur, il est judicieux d'opter pour des types énumérés(3.4.2).

Voici la version en Java, une version similaire en Python sera donnée en dessous.

---

**Algorithme 8.1** Les types énumérés utilisés pour définir un carte d'un jeu de cartes

---

```
//Les 3 enum seront encodés dans 3 fichiers différents
//Il est cependant possible de les mettre directement dans le record "Carte"
public enum Couleur {NOIR, ROUGE}

public enum Motif {
    AS, DEUX, TROIS, QUATRE, CINQ, SIX, SEPT, HUIT, NEUF, DIX,
    VALET, DAME, ROI
}
//une version plus élaborée permettrait de préciser directement la couleur au sein
// de la Forme, pour ne pas complexifier le code, cette version ne sera pas choisie
// ici
public enum Forme {PIQUE, TREFLE, CARREAU, COEUR}
```

---

On va maintenant créer l'enregistrement « Carte » de la façon suivante:

---

**Algorithme 8.2** Création d'un enregistrement « Carte »

---

```
public record Carte(int valeur, Motif motif, Forme forme, Couleur couleur) {}
//Exemple de création d'une carte
Carte dameCoeur= new Carte(12,Motif.DAME,Forme.COEUR,Couleur.ROUGE);
System.out.println(dameCoeur);
```

```
Carte[valeur=12, motif=DAME, forme=COEUR, couleur=ROUGE]
```

---

La version Python donnerait ceci:

---

**Algorithme 8.3** Version Python d'une carte

---

```
from enum import Enum
class Couleur(Enum):
    NOIR=0
    ROUGE=1
class Forme(Enum):
    PIQUE=0
    TREFLE=1
    CARREAU=2
    COEUR=3
class Motif(Enum):
    AS=0
    DEUX=1
    TROIS=2
    QUATRE=3
    CINQ=4
    SIX=5
    SEPT=6
    HUIT=7
    NEUF=8
    DIX=9
    VALET=10
    DAME=11
    ROI=12

dameCoeur = (12,Motif.DAME, Forme.COEUR,Couleur.ROUGE)
print(dameCoeur)

(12, <Motif.DAME: 12>, <Forme.COEUR: 3>, <Couleur.ROUGE: 1>)
```

---

## Les listes, les ensembles et les dictionnaires en Java

Après avoir consacré le chapitre précédent à Python, nous allons voir que Java n'est pas reste car le langage offre plusieurs classes pour gérer des listes, des ensembles, des dictionnaires,... Cependant le fonctionnement sera plus classique qu'en Python car il s'agit de classes que l'on devra appeler via des méthodes pour les construire et les manipuler, il n'y aura pas d'opérateur qui permettront de créer directement ces structures.

Le langage Java utilise la notion d'interface que j'appelle généralement un contrat. Une interface ne contient que des méthodes abstraites<sup>1</sup> D'autre part, plusieurs classes pourront implémenter une interface pour fournir plusieurs versions de ce contrat. Cette façon de procéder est très intéressante pour d'une part faciliter le développement en permettant de séparer le développement du code qui utilise l'interface et celui qui l'implémente. D'autre part, le fait d'avoir plusieurs implémentations permettra de choisir celle qui sera la plus performante pour ses besoins.

Regardons quelques aspects qui distinguent les principales structures de Java:

Structure	Description	mutable *	ordre	doublon
Array	vecteur (sans méthode)	oui	oui	oui
Collections	liste d'objets	oui	non	oui
List	liste d'objets	oui	oui	oui
Set	Ensemble d'objets	oui	non **	non
Map	Dictionnaire clé => valeur	oui	non**	—

\*: Il est possible de rendre ou de construire des structures qui sont extensibles ou non, modifiables ou non. Par défaut elles sont mutables.

\*\*: Cela dépendra de l'implémentation.

TABLEAU 9.1 – Les structures en Java

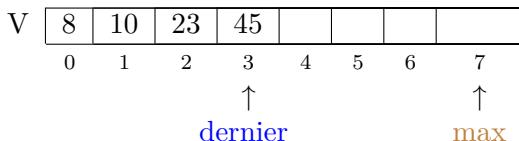
### 9.1 La notion d'interface et d'implémentation

En Java, le langage est très souvent construit en distinguant d'une part la définition d'une interface qui contient les signatures des méthodes et d'autre part les classes qui implémentent l'interface (le

1. Dans les versions actuelles de Java, une méthode d'une interface, peut contenir une implémentation par défaut. Une interface peut également définir des constantes ou des méthodes statiques.

contrat). L'intérêt de faire ainsi est de pouvoir changer très facilement d'implémentation pour une même interface. Prenons l'exemple d'une liste, il existe une interface « *List* » qui contient toutes les signatures des méthodes indiquant ce qu'on peut faire sur une liste. Mais l'implémentation d'une liste, peut se réaliser avec un vecteur ou avec une liste chaînée, ces deux techniques ont chacune des avantages et des inconvénients. D'autre part, si la liste doit être accessible par plusieurs processus, l'implémentation sera plus complexe à mettre en œuvre. Regardons brièvement les avantages et inconvénients d'un vecteur et d'une liste chainée:

### 1. utilisation d'un vecteur, avantages et inconvénients



- ☞ si on a besoin d'accéder aux éléments via leurs indices « *v[i]* »
- ☞ les insertions en fin de liste seront rapides du moins si elle ne dépasse pas la taille maximum
- ☞ une insertion à l'intérieur du vecteur sera plus lente car elle nécessite un décalage des autres valeurs
- ☞ si on dépasse la taille maximum du vecteur, il faudra recopier le vecteur existant dans un nouveau vecteur plus grand

### 2. utilisation d'une liste chainée, avantages et inconvénients

chaine → → → → →

- ☞ intéressant lorsque l'on doit effectuer beaucoup d'insertions à l'intérieur du vecteur car il suffit de créer un nouvel élément et d'ajuster les pointeurs
- ☞ si la taille change souvent et n'est pas connue au départ, on peut facilement chainer de nouveaux éléments
- ☞ l'accès à un élément via sa position est très lente car il faudra parcourir la chaîne jusqu'à l'élément
- ☞ une liste chaînée est plus complexe à gérer car chaque élément doit être un bloc mémoire qui doit aussi contenir un pointeur vers l'élément suivant
- ☞ ne permet pas de circuler dans le sens inverse, à moins qu'on utilise une liste doublement chaînée mais sa gestion est plus lourde et plus coûteuse

Voici un exemple de création d'une liste:

---

#### Algorithme 9.1 Interface et implémentation

---

```
List<Integer> liste= new ArrayList<>();
```

Interface "List"

Le type de ses éléments  
(Objets)

L'implémentation choisie  
(vecteur)

Taille initiale par défaut.  
Type, idem à l'interface

**List<Integer> liste= new ArrayList<>();**

---

L'exemple ci-dessus déclare une variable « *liste* » du type de l'interface « *List* », cette interface est une interface générique càd qu'elle nécessite un paramètre qui précise le type des éléments de la liste, ce type doit être un type objet et pas un type primitif. Ensuite il faut assigner une implémentation

pour cette interface, ici j'ai opté pour « *ArrayList* » qui comme son nom l'indique, utilise un vecteur pour implémenter la liste. Par défaut le vecteur aura une taille de 10 éléments mais si vous savez au départ qu'elle fera plus de 10 éléments, vous pouvez préciser une autre taille initiale. Bien évidemment, il sera possible de mettre plus de 10 éléments dans la liste mais à partir du 11<sup>ème</sup> élément la classe *ArrayList* devra créer un nouveau vecteur plus grand.

En dehors des classes et interfaces que nous allons voir dans les prochains points, il faudra aussi consulter les classes utilitaires suivantes:

**Collections** Cette classe se compose exclusivement de méthodes statiques qui fonctionnent sur ou renvoient des collections. Elle possède plusieurs méthodes pour inverser, faire une rotation, mélanger les éléments d'une liste ainsi que des méthodes pour rendre une liste, un ensemble ou un dictionnaire synchronisée ou immuable.

**Arrays** Cette classe contient diverses méthodes de manipulation de tableaux (telles que le tri, la recherche, la copie, le test d'égalité, le remplissage,...). Cette classe contient également une fabrique statique qui permet de visualiser les tableaux sous forme de listes.

## 9.2 Les listes en Java

L'interface « *List* » définit toutes les méthodes que chaque implémentation devra définir. Les principales implémentation de cette interface sont:

**ArrayList<E>** implémentation via un vecteur

**LinkedList<E>** implémentation via une liste chaînée

**Vector<E>** implémentation d'un vecteur mais accessible par plusieurs *threads* car ses méthodes sont synchronisées.

**Stack<E>** étend la classe *Vector* en fournissant en plus les méthodes *push*, *pop* et *peek*.

Dans le cadre de ce cours, nous allons nous contenter de la classe « *ArrayList* » pour implémenter le contrat « *List* », il s'agit d'un choix assez fréquent.

### 9.2.1 Crédation et initialisation d'une liste

Java fonctionne de manière assez traditionnelle comme langage orienté objet, on crée un objet en faisant appel à un constructeur et on assigne cet objet à un référent qui sera dans notre cas du type de l'interface « *List* ». On peut aussi définir une liste avec des valeurs initiales à partir d'une méthode statique mais la liste ne sera plus modifiable dans ce cas:

#### Création d'une liste initialisée et non modifiable

```
//Liste non modifiable
List<Integer> v1=List.of(2, 3, 5, 7, 11, 13, 17, 23);
```

#### Création d'une liste initialisée, modifiable mais de taille fixe (non extensible)

```
//Liste modifiable mais de taille fixe
List<Integer> v2= Arrays.asList(1,2,5,6);
v2.set(0,10); //Insère l'entier 10 en position 0
System.out.println(v2); //Affichera: [10, 2, 5, 6]
```

### Création d'une liste modifiable

```
//Liste vide extensible (taille initiale 10)
List<Integer> v3= new ArrayList<>(); //La taille initiale peut être précisée via le
→ constructeur
//Ajout d'une liste d'entiers
v3.addAll(Arrays.asList(21,33,2,9));
System.out.println(v3); //Affichera: [21, 33, 2, 9]
```

### 9.2.2 Principales méthodes de manipulation d'une liste:

v	10	4	-3	9
indices:	0	1	2	3

Chaque exemple du tableau ci-dessous, s'exécute à partir de la liste « v »:

Opérations	Code Java	Résultat
taille d'une liste	v.size();	4
accès à un élément en position i	v.get(0);	10
remplacer un élément en position i	v.set(0, 55);	[55, 4, -3, 9]
intercaler un élément à une position	v.add(1, 99);	[10, 99, 4, -3, 9]
insère un élément en fin de liste	v.add(22);	[10, 4, -3, 9, 22]
intercaler plusieurs éléments à une position	v.addAll(1,Arrays.asList(1,2,3));	[10, 1, 2, 3, 4, -3, 9]
insère plusieurs éléments en fin de liste	v.addAll(Arrays.asList(1,2,3));	[10, 4, -3, 9, 1, 2, 3]
supprimer un élément en position 2	v.remove(2);	[10, 4, 9]
supprimer la <u>première occurrence</u> d'un élément	v.remove((Integer) (-3));	[10, 4, 9]
supprime tous les éléments qui répondent à un prédictat	v.removeIf(x -> x<5);	[10, 9]
est-ce que la liste contient l'élément 9?	v.contains(9);	true
vide la liste	v.clear();	[]
inverser une liste	Collections.reverse(v);	[9, -3, 4, 10]
trier une liste avec le comparateur de la classe <i>Integer</i>	v.sort(Integer::compare);	[-3, 4, 9, 10]
trier une liste avec le comparateur par défaut	Collections.sort(v);	[-3, 4, 9, 10]
envoie une partie de liste de l'index de début, à l'index de fin non compris	v2=v.subList(1, 3);	[4, 9]

TABLEAU 9.2 – Méthodes de manipulation d'une liste

### 9.2.3 Parcourir une liste

Pour parcourir une liste, on peut utiliser les mêmes techniques qu'avec les vecteurs, càd en utilisant la boucle « *for* » classique ou sa version « *For Each* » ou encore l'utilisation d'un *itérateur*:

---

**Algorithme 9.2** Parcourir une liste Java

---

```
// Parcourir une liste avec une boucle "pour"
for (int i = 0; i < v.size(); i++)
    System.out.print(v.get(i) + " , ");
System.out.println();

// Parcourir avec une boucle "For each"
for (Integer elem : v)
    System.out.print(elem + " , ");
System.out.println();

//Avec un itérateur
var iter=v.iterator();
while (iter.hasNext())
    System.out.print(iter.next() + " , ");
System.out.println();
```

---

## 9.3 Les ensembles en Java

Un ensemble est une structure de mémorisation sans doublon et sans ordonnancement (sauf pour certaines implémentations).

L'interface pour définir un ensemble est « `Set<E>` » et ses principales implémentations sont:

**HashSet<E>** utilise une table de *hashing*. L'ordre de ses éléments n'est pas garanti ni constant. L'élément *null* est autorisé. Les méthodes ne sont pas synchronisées.

**TreeSet<E>** utilise une *TreeMap* pour implémenter l'ensemble, les éléments sont ordonnés selon l'ordre d'ajout ou selon un comparateur spécifié dans le constructeur. L'élément *null* n'est pas autorisé. Cette structure n'est pas synchronisée.

### 9.3.1 Crédation et initialisation d'un ensemble

Java fonctionne de manière assez traditionnelle comme langage orienté objet, on crée un objet en faisant appel à un constructeur et on assigne cet objet à un référent qui sera dans notre cas du type de l'interface « `Set` ». On peut aussi définir une ensemble immuable à partir d'une méthode statique « `Set.of` »:

#### Création d'un ensemble initialisé et non modifiable

```
//Ensemble non modifiable
Set<Integer> ens1=Set.of(1,2,3,5,8,13);
```

#### Création d'un ensemble modifiable

```
//Ensemble vide extensible (taille initiale 16 et un "load factor" de 0.75)
Set<Integer> e1=new HashSet<>();//La taille initiale et lf peuvent être précisés via
→ le constructeur
//Ensemble modifiable initialisé à partir d'une collection
Set<Integer> e2=new HashSet<>(Arrays.asList(3,6,9,12,15));
System.out.println(e2)//Affichera: [3, 6, 9, 12, 15]
```

Le « *loadfactor* » est par défaut à 0.75, il signifie qu'à 75% de remplissage de l'ensemble, le vecteur de *hashing* interne sera étendu et un *rehashing* des données sera effectué. Si on connaît la taille maximale de l'ensemble à créer, il est intéressant de la définir à l'avance en tenant compte du facteur « *lf* » ainsi pour 100 éléments on définira une taille de minimum  $\lceil 100/0.75 \rceil = 134$ .

### 9.3.2 Principales méthodes de manipulation sur un ensemble:

e1	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>5</td><td>8</td><td>13</td></tr></table>	1	2	3	5	8	13
1	2	3	5	8	13		
e2	<table border="1"><tr><td>1</td><td>2</td><td>4</td><td>8</td><td>16</td><td>32</td></tr></table>	1	2	4	8	16	32
1	2	4	8	16	32		
Opérations	Code Java	Résultat					
taille	e1.size();	6					
ajouter un élément à l'ensemble	e1.add(21); e1.add(2)	[1, 2, 3, 5, 8, 13, 21]					
ajoute plusieurs éléments	e2.addAll(Arrays.asList( 64, 128));	[1, 2, 4, 8, 32, 16, 32, 64, 128]					
supprimer l'élément de l'ensemble	e2.remove(16);	[1, 2, 4, 8, 32]					
les éléments qui répondent à un prédictat	e1.removeIf(x -> x%2==0);	[1, 3, 5, 13]					
contient un élément	e1.contains(3);	true					
vide l'ensemble	e1.clear();	[]					
garde l'intersection entre 2 ensembles	e1.retainAll(e2);	[1, 2, 8]					
garde les éléments qui n'existent pas dans l'autre ensemble	e1.removeAll(e2);	[3, 5, 13]					
Retourne un <i>Array</i> avec tous les éléments du vecteur	Integer[] v=e1.toArray();	v = [1, 2, 3, 5, 8, 13]					

TABLEAU 9.3 – Méthodes de manipulation sur un ensemble

### 9.3.3 Parcourir les éléments d'un ensemble

Pour parcourir un ensemble, on peut utiliser la boucle « *For Each* » ou un *itérateur*. L'ordre n'est pas garanti sauf si l'implémentation le permet.

---

**Algorithme 9.3** Parcourir les éléments d'un ensemble Java
 

---

```
// Parcourir un ensemble avec une boucle "For each"
for (Integer e: e2)
    System.out.print(e+",");
System.out.println();

//Une boucle For Each sous forme d'une lambda expression
e2.forEach(e -> System.out.print(e + ","));
System.out.println();

// Avec un itérateur
var iter1 = e2.iterator();
while (iter1.hasNext())
    System.out.print(iter1.next() + ",");
System.out.println();
```

---

## 9.4 Les dictionnaires en Java

Un dictionnaire est une structure *clé → valeur* où la clé est un objet unique selon les méthodes *equals* et *hashCode* et la valeur est également un objet. L'interface à implémenter est « *Map< K, V >* » où *K* désigne le type des objets clés et *V* le type des objets valeurs.

Les principales implémentations de l'interface *Map* sont:

**HashMap<K,V>** cette implémentation est très proche d'une *HashTable* à l'exception qu'elle n'est pas synchronisée et qu'elle accepte *null* aussi bien pour une clé que pour une valeur.

**Hashtable<K,V>** n'importe quel objet non nul est autorisé comme clé ou comme valeur. Ses méthodes sont synchronisées.

**Properties** Les clés et les valeurs sont du type *String*. Cette implémentation est spécifique pour charger ou sauver une liste de propriétés à partir ou vers un flux.

Dans notre cas, on optera principalement pour l'implémentation *HashMap* car elle est performante et nous n'avons pas besoin d'une implémentation synchronisée. Via son constructeur, il est possible de préciser sa capacité qui définit son nombre d'emplacements et son *loadFactor*, qui est par défaut à 0.75, il signifie qu'à partir de 75% d'occupation, le nombre d'emplacements sera augmenté.

### 9.4.1 Crédation et initialisation d'un dictionnaire

L'interface à implémenter est « *Map* ». On peut aussi définir un dictionnaire non modifiable à partir de la méthode statique « *Map.of* »:

#### Création d'un dictionnaire initialisé et non modifiable

```
//Dictionnaire non modifiable
//Map non modifiable v1
Map<Integer, String> codesHttp1= Map.of(200, "Succès", 301, "Indirection", 404, "Ressource
→ non trouvée");
//Map non modifiable v2
Map<Integer, String> codesHttp2= Map.ofEntries(Map.entry(200, "Succès"),
→ Map.entry(301, "Indirection"), Map.entry(404, "Ressource non trouvée"));
```

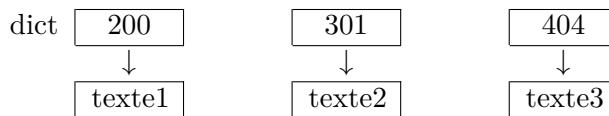
---

### Création d'un dictionnaire modifiable

```
//Dictionnaire vide extensible (taille initiale 16 et un "load factor" de 0.75)
Map<Integer, String> codesHttp3=new HashMap<>(); //La taille initiale et lf peuvent
→ être précisés via le constructeur
//Ajout d'entrées
codesHttp3.put(200, "Succès");
codesHttp3.put(301, "Indirection");
codesHttp3.put(404, "Ressource non trouvée");
```

Le « *loadfactor* » est par défaut à 0.75, il signifie qu'à 75% de remplissage, le nombre d'entrées sera augmenté et un *rehashing* des clés sera effectué. Si on connaît le nombre d'entrées maximum du dictionnaire, il serait intéressant de définir sa capacité via le constructeur en tenant compte du facteur « *lf* », ainsi pour 100 éléments on définira une taille de minimum  $\lceil 100/0.75 \rceil = 134$ .

### 9.4.2 Principales méthodes applicables sur un dictionnaire:



Opérations	Code Java	Résultat
taille	dict.size();	3
ajouter un élément	dict.put(302, "texte22")	
obtenir un élément à partir de la clé	dict.get(404); dict.getOrDefault(405, "txtDéfaut")	« texte3 » « txtDéfaut »
supprimer l'élément à partir de sa clé	dict.remove(404); dict.remove(404, "texte3")	« texte3 » true
supprime l'élément à partir de sa clé et sa valeur		
remplace un élément	dict.replace(404, "texte4")	« texte3 »
contient une clé	dict.containsKey(404);	true
contient une valeur	dict.containsValue("texte3")	true
vide le dictionnaire	dict.clear();	
ensemble des clés	dict.keySet()	[200, 301, 404]
ensemble des entrées	dict.entrySet()	les paires: clé-valeur
collection des valeurs	dict.values()	["texte1", "texte2", "texte3"]

TABLEAU 9.4 – Méthodes de manipulation sur un dictionnaire

### 9.4.3 Parcourir les éléments d'un dictionnaire

Sur un dictionnaire, il est possible d'avoir:

- ☛ l'ensemble des clés
- ☛ la collection des valeurs (objets)
- ☛ l'ensemble des entrées (Clé, Valeurs)

Chacun de ces éléments, peut être parcouru à l'aide de la boucle « *For Each* » ou avec un *itérateur*. L'ordre n'est pas garanti sauf si l'implémentation le permet.

---

#### **Algorithme 9.4** Obtenir les éléments d'un dictionnaire

---

```
//Ensemble des clés
System.out.println(codesHttp3.keySet());
//Affichera [404, 200, 301]

//Ensemble des entrées
System.out.println(codesHttp3.entrySet());
//Affichera: [404=Ressource non trouvée, 200=Succès, 301=Indirection]
//Collection des valeurs (objets)
System.out.println(codesHttp3.values());
//Affichera: [Ressource non trouvée, Succès, Indirection]
```

---



# Chapitre 10

## Les chaînes de caractères

Dans tous les langages de programmation, vous trouverez des variables qui permettent de stocker un texte. Un texte est en réalité une suite de caractères qui sera traité d'une manière spécifique.

//TODO

### 10.1 *Parser du texte*



# Chapitre 11

## Les procédures et fonctions

Pour éviter de répéter plusieurs fois un même code de programmation, on va pouvoir le mettre au sein d'une procédure ou d'une fonction, celui-ci pourra par la suite être appelé autant de fois que nécessaire, tout en ayant la possibilité de lui fournir des paramètres d'entrée différents. Quelle est la différence entre une fonction et une procédure? Certains langages font une distinction entre les 2 en ayant une syntaxe distincte, d'autres langages comme Java, C++, Python utilisent une même syntaxe de programmation. Une fonction retourne une valeur de telle sorte qu'elle puisse faire partie d'une expression alors qu'une procédure ne retourne aucune valeur, elle peut cependant modifier des variables fournies via ses paramètres d'entrée.

**La notion de fonction:** Imaginons un bloc de code qui calcule  $x^y$ , appelons le « `power(x,y)` », cette fonction nécessite 2 paramètres d'entrée x et y, en résultat elle renverra une valeur qui sera égale à  $x^y$ . On a affaire dans ce cas à une **fonction** car elle retourne une valeur qui peut directement être exploitée dans une expression, on peut substituer la fonction par sa valeur de retour, au même titre que  $\sin(x)$ ,  $\cos(x)$ ,  $\min(x,y)$ ,...

Voici un exemple d'expression qui appelle les fonctions *sinus* et *power*:  $w = 1 + \sin(x) * \text{power}(x, y)$

### Fonction

Une fonction est un bloc de code auquel on peut fournir des paramètres et qui renvoie une information. Elle peut donc être intégrée à une expression.

Le retour d'une fonction est défini par l'instruction « `return` »!

Voici un exemple de fonction, écrite en Java et en Python, qui retourne la valeur minimum entre 2 nombres:

---

**Algorithme 11.1** Exemple de fonction

---



```

def minValue(a,b):
    #Retour de la fonction
    return a if a<=b else b
#Exemple d'appel
a=...
b=...
res = a + minValue(a,b)#Appel dans une expression
...
//fonction qui retourne la valeur minimum entre 2 nombres
public static int minValue(int a,int b) {
    //Retour de la fonction
    return a<=b? a: b;
}
//Exemple d'appel de la fonction:
public static void main(String[] args){
    int a,b;
    ...
    int res=a + minValue(a,b); //appel dans une expression
    ...
}

```

---

**La notion de procédure** Une procédure est également un bloc de code que l'on peut appeler mais qui ne retourne aucune valeur et que l'on ne peut pas, de ce fait, intégrer à une expression. En Java une procédure n'est rien d'autre qu'une fonction qui ne retourne rien, dont le type de retour sera « `void` », en Java comme en Python une procédure n'aura pas d'instruction « `return` »!

---

**Algorithme 11.2** Exemple de procédure

---



```

//affiche les valeurs de a à b
def affiche(a, b):
    for i in range(a, b+1):
        print(i, end=' ')
...
#Exemple d'appel
affiche(2,8)#exécute la procédure
...

//affiche les valeurs de a à b
public static void affiche(int a,int b) {
    for (int i=a; i<=b; i++)
        System.out.print(i+" ");
}
//Exemple d'appel d'une procédure:
public static void main(String[] args){
    ...
    //Appel de la procédure
    affiche(2,8);
    ...
}

```

---

## Procédure

Une procédure est un bloc de code auquel on peut fournir des paramètres mais qui ne retourne aucune donnée.

## 11.1 Le passage de paramètres

Commençons par quelques mots de vocabulaire:

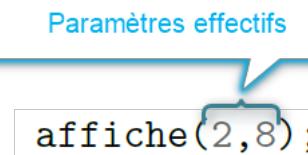
- les arguments précisés dans l'entête de la fonction sont les **paramètres formels**:



```
public static void affiche(int a,int b)
```

A chaque appel de la fonction, les paramètres formels créeront des variables locales stockées sur le stack, elles seront initialisées par les valeurs correspondantes précisées par les paramètres effectifs.

- les arguments précisés lors de l'appel de la fonction sont appelés les **paramètres effectifs**:



```
affiche(2,8);
```

Un paramètre effectif peut être une expression, une variable de type primitif, un objet mutable ou un objet immuable.

Nous verrons qu'en fonction des langages, il existe différentes façons de préciser les paramètres formels et donc de faire correspondre les paramètres effectifs aux paramètres formels.

### 11.1.1 Le type des arguments

Nous avons déjà souligné le fait qu'à chaque appel d'une fonction les paramètres formels deviendront des variables locales propres à l'appel de la fonction et uniquement visibles dans celle-ci. Ainsi à chaque appel de la fonction, de nouvelles variables locales seront créées sur le *stack*.

Une autre chose très importante à comprendre, c'est l'implication d'une modification d'un paramètre formel sur les variables externes à la fonction, celles précisées lors de l'appel.

Étudions les différents cas que l'on pourrait rencontrer:

Nous allons regardons en détail les 4 cas suivants où les types des paramètres effectifs sont:

- une expression où une variable avec un type primitif :
- un objet immuable (non modifiable) sans méthode de modification
- un objet immuable avec des méthodes de modification
- un objet mutable

## Paramètre formel ayant un type primitif

Si l'argument formel est d'un type primitif, on parle d'[envoi par copie](#), c'd que l'information précisée lors de l'appel sera recopiée dans la variable locale du paramètre formel. Ainsi la modification de la variable restera locale à la fonction. On parle bien ici de tout type primitif qui n'est pas une adresse (pointeur ou référent)!

Dans l'exemple suivant à l'appel de la fonction « `affiche(x,y)` » les valeurs de « `x` » et « `y` » seront recopierées dans les variables locales « `a` » et « `b` » et la modification de la variable « `a` » au sein de la fonction, n'influencera pas la variable extérieure « `x` ».

```
//les arguments a et b sont du type primitif "int"
public static void affiche(int a,int b) {
    //La modification de "a" n'influencera pas les variables précisées
    // lors de l'appel
    for (; a<=b; a++)
        System.out.print(a+" ");
}

//Exemple d'appel d'une procédure:
public static void main(String[] args){
    ...
    //Appel de la procedure
    int x=4;
    int y=8;
    affiche(x,y);
    //x et y auront toujours la même valeur
    ...
}
```

Il est ainsi possible de préciser une expression dans un paramètre effectif, exemple:

```
affiche(x-2,y+5)
```

## Paramètre formel d'un type objet (référent).

Quand le paramètre effectif est un objet, c'est l'adresse de l'objet qui sera recopiée dans le paramètre formel. Ainsi à l'appel, il faudra obligatoirement avoir un objet et non pas une expression. Maintenant, est-ce-qu'il y a moyen de modifier le paramètre effectif à partir du paramètre formel? Tout cela va dépendre de l'objet envoyé, s'il est modifiable ou non.

**Objet immuable** Si l'objet est immuable(14.5), donc non modifiable. La question ne se pose pas. Le paramètre effectif désignera bien le même objet que celui précisé lors de l'appel de la fonction mais comme il n'est pas modifiable, il n'y a pas de risque de modification.

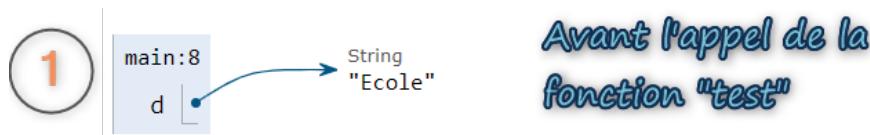
**Objet immuable ayant des méthodes de modification** En Java les classes `String`, `Integer`, `Color`,... en Python, les classes `Integer`, `Float`, `Boolean`, `String` et `Tuple` sont immuables cependant sur certaines classes, on peut y retrouver des méthodes pour « modifier » l'objet, cependant ces méthodes ne modifient pas l'objet mais en renvoie un nouveau!

Ainsi dans l'exemple suivant:

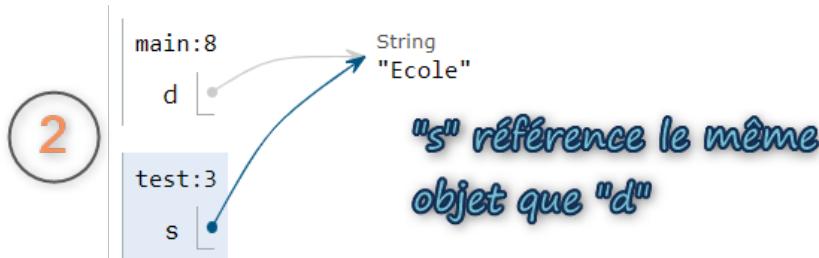
```

1  public class Exemple {
2      //Fonction avec un paramètre d'un type immuable
3      public static String test(String s){
4          s=s.toUpperCase();
5          return s;
6      }
7      public static void main(String[] args) {
8          String d="Ecole";
9          test(d); //Appel de la fonction
10         System.out.println(d);
11     }
12 }
```

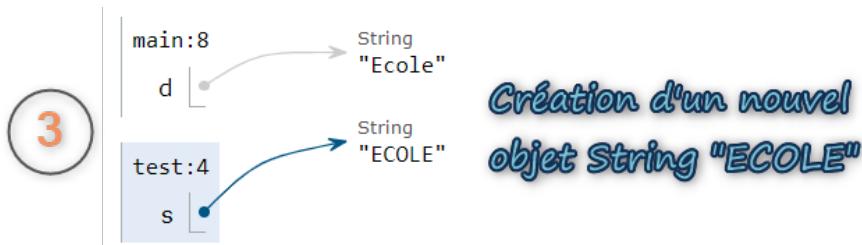
1. voici l'état avant l'appel de la fonction « test »:



2. quand on entre dans la fonction « test », le paramètre formel pointe bien vers le texte « Ecole »:



3. quand on modifie la variable « s », on constate la création d'un nouvel objet, ce qui met bien en évidence que l'objet de type String est immuable:



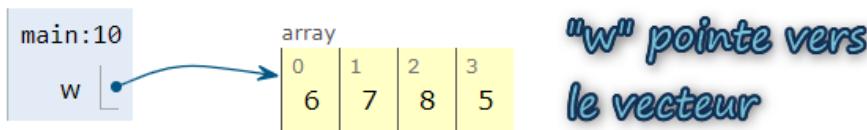
**Paramètre effectif avec un objet mutable** Si un paramètre formel reçoit un objet mutable, alors la fonction pourra modifier cet objet. Imaginons que l'on fourni un vecteur à une fonction, comme un vecteur est modifiable, la fonction pourra si nécessaire modifier l'objet.

Prenons comme exemple une procédure qui va faire une rotation droite des éléments d'un vecteur:

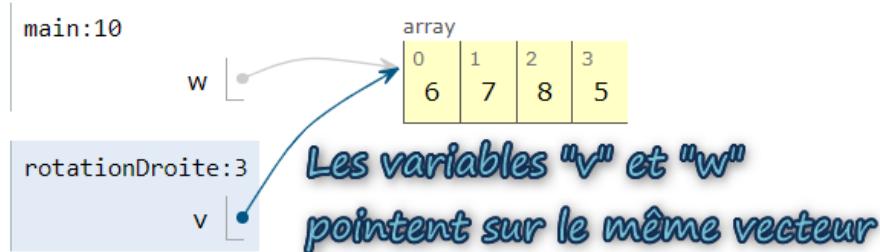
```

1 public class Exemple {
2     public static void rotationDroite(int[] v){
3         int tmp= v[v.length-1];
4         for (int i=v.length-1;i>0;i--)
5             v[i]=v[i-1];
6         v[0]=tmp;
7     }
8     public static void main(String[] args) {
9         int[] w={6,7,8,5};
10        rotationDroite(w);
11        // "w" a été modifié par la procédure
12        System.out.println(w);
13    }
14 }
```

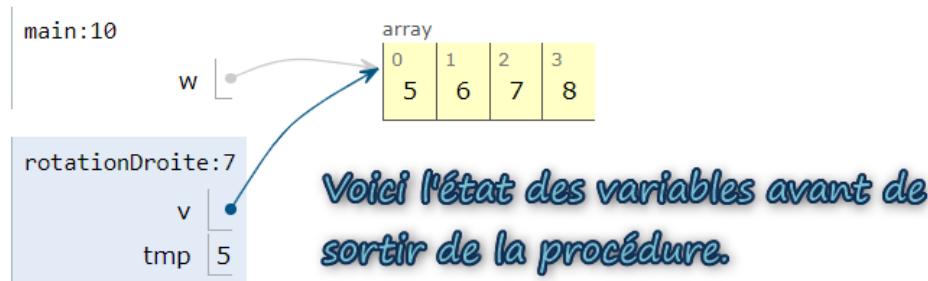
- avant l'appel de la procédure, la variable « w » pointe vers le vecteur:



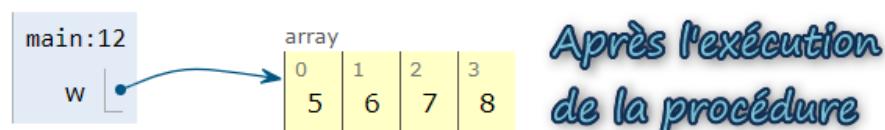
- à l'entrée de la procédure le paramètre formel « v » pointe maintenant sur le même vecteur que la variable externe « w »:



- lorsque je modifie dans la procédure le vecteur « v », on peut constater que c'est le même vecteur que l'on modifie, ainsi la variable « w » aura bien accès au vecteur modifié car un vecteur est un objet mutable!



- après l'appel de la procédure, les variables créées dans la procédure ont été supprimées et « w » pointe bien vers le vecteur modifié:



## 11.2 Créez des fonctions et procédures en Java et en Python

Nous avons vu précédemment les notions de fonction et de procédure, dorénavant nous ne ferons plus la distinction en les deux car la structure est identique aussi bien en Java qu'en Python, bien entendu une fonction aura une valeur de retour précisée via l'instruction « *return* » alors qu'une procédure n'en aura pas et aura un type de retour « *void* » (en Java).

Nous avons également vu précédemment le comportement du passage de paramètres et ceci en fonction que l'on envoie un type primitif, un objet mutable et un objet immuable. Maintenant nous allons regardons les différentes façons de définir les arguments dans les 2 langages étudiés. Commençons par le langage Java, suivi par le langage Python qui offre beaucoup plus de possibilités.

### 11.2.1 Définition d'une fonction en Java

L'entête d'une fonction désigne ce qu'on appelle la **signature de la fonction**, il est possible d'avoir plusieurs fonctions ayant le même nom mais avec une liste d'arguments différents en nombre ou en type, de telle sorte qu'il n'y ai pas de confusion possible lors de l'appel de la fonction.

En java, comme toute fonction se retrouve au sein d'une classe, on parle plus précisément de **méthode** :

**méthode d'objet:** c'est une fonction qui s'applique sur des objets, on ne peut pas l'appeler sans avoir d'objet. L'appel se fera à partir d'un objet: « *objet.méthode(param1,param2)* ». La méthode aura un paramètre caché « *this* » qui désignera l'objet, elle pourra ainsi avoir accès aux données de l'objet. Dans ce cours nous ne ferons pas, voire très peu de méthodes d'objet, car la programmation orientée objet ne fait pas partie du programme de cette unité d'enseignement.

**méthode de classe:** une méthode de classe est une fonction que l'on peut appeler à partir de la classe, elle n'a pas besoin d'objet pour être appelée, on l'appelle via le nom de la classe. On parle dans ce cas d'une **méthode statique**. La plupart des méthodes créées dans ce cours seront des méthodes statiques.

Commençons par regarder les différents éléments que l'on retrouve généralement dans une méthode Java:

```
/**  
 * Calcul le PGCD selon Euclide  
 * @param a un entier >0  
 * @param b un entier >0  
 * @return le pgcd de a et b  
 */  
  
public static int pgcd(int a, int b){  
    assert a > 0 && b > 0 : "a et b doivent être >0";  
    int reste;  
    while (b != 0) {  
        reste = a % b;  
        a = b;  
        b = reste;  
    }  
    return a;  
}
```

1. un commentaire *JavaDoc* qui permet de décrire la fonction et ses paramètres via des tags « *@param*, *@return*,... », il permettra d'afficher l'aide sur une fonction à partir d'une IDE et d'avoir la description de la fonction dans la documentation *JavaDoc* de la classe .

2. l'entête de la méthode dont le format sera détaillé en dessous
  
  
  
3. L'assertion est une condition qui doit être validée pour poursuivre l'exécution. Ceci permet de debugger plus facilement un programme en précisant à chaque fonction ses hypothèses d'entrée. Les assertions ont l'avantage de pouvoir être activées ou non, pour les activer, il faut rajouter dans les arguments de la VM<sup>1</sup> le paramètre « `-ea` » pour « *Enable Assertion* »:



4. le corps de la méthode entre 2 accolades
  
  
  
5. l'instruction « *return* » pour préciser le retour de la fonction. Dans le cas d'une procédure, le retour sera à « *void* » et il n'y aura pas de présence d'instruction « *return* ».

Regardons maintenant comment définir les arguments d'une fonction en commençant par la manière classique suivie par la structure *VarArgs* qui permet de définir un nombre variable d'arguments.

Java n'a pas la possibilité de fournir des valeurs par défaut pour ses paramètres formels, ainsi lors de l'appel d'une fonction, chaque argument formel devra correspondre à un argument effectif.

#### **Envoi classique des arguments:**

Une fonction peut avoir de 0 à n arguments, chaque argument devra être précisé avec son type suivi de son nom.

---

1. VM: *Virtual Machine*, la machine virtuelle de Java

**Algorithme 11.3** Envoi classique des arguments en Java

```

public class Exemples {

    //fonction sans argument
    public static String exemple1(){
        return "==== Exemple 1 ====";
    }

    //fonction avec deux arguments, un vecteur d'entiers et une valeur
    public static int nbVal(int[] v,int val) {
        int cpt=0;
        for(int elem : v)
            if (elem == val) cpt++;
        return cpt;
    }

    //méthode "main" avec un vecteur de String en argument
    public static void main(String[] args) {
        //appel de la fonction sans argument
        System.out.println(exemple1());

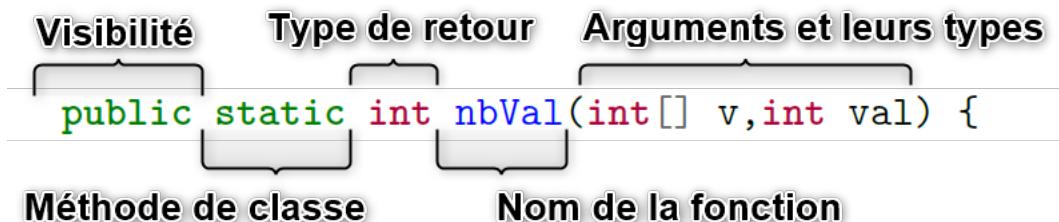
        int[] w={6,5,8,5,9};
        //appel de la fonction nbVal
        int res= nbVal(w,5);
        //affichera 2 car "w" contient 2 fois la valeur 5
        System.out.println(res);
    }
}

```

==== Exemple 1 ====  
2

Résultat dans la console:

Une fonction est donc composée de plusieurs parties:



- la visibilité de la méthode qui sera dans notre cas souvent « `public` », mais on peut aussi avoir les visibilités « `private` » et « `protected` ».
- « `static` » qui permet d'indiquer que la méthode est une méthode de classe
- le type de retour ou « `void` » si la fonction ne retourne rien
- le nom de la fonction au format « *camelCase* » mais en commençant par une minuscule
- rien ou une liste d'arguments sous le format: « `type nomArg` »
- le corps de la fonction entre 2 accolades

## Envoi via une liste variable d'arguments

Java possède une notation qui permet d'appeler une fonction avec un nombre variable d'arguments de même type (type compatible)

Voici une fonction qui calcule la somme de ses arguments:

---

### Algorithme 11.4 Envoi d'un nombre variable d'arguments en Java

---

```
public class Exemple2 {
    //fonction avec un argument qui sera un vecteur d'entiers
    public static int somme(int... v){
        int s=0;
        //tous les arguments sont dans un vecteur comme: int[] v
        for (int elem : v)
            s = s + elem;
        return s;
    }

    public static void main(String[] args) {
        //appel de la fonction avec 3 arguments
        int res1= somme(3,7,9);
        System.out.println("res1 = "+res1);
        //appel de la fonction avec 8 arguments
        int res2= somme(1,8,5,-1,7,3,2,9);
        System.out.println("res2 = "+res2);
    }
}
```

Résultat dans la console: res1 = 19  
res2 = 34

---

Si la fonction possède plusieurs paramètres formels, le paramètre variable doit être en dernière position de telle sorte qu'aucune confusion ne soit possible lors de l'appel:

*En dernière position*



```
public static int somme(int a,int... v){
```

## 11.2.2 Définition d'une fonction en Python

Commençons par regarder la structure classique d'une fonction en Python, avec l'exemple suivant qui calcule la factorielle de n:

```
1 def factorielle(n): Signature de la fonction avec son nom et ses paramètres formels
2     ''' Calcule la factorielle de n ''' Description de la fonction
3     assert type(n)== int and n>=0, "n doit être un entier positif" Assertion pour le débug, peut-être désactivée avec l'option "-O"
4     y=1
5     for i in range(2,n+1):
6         y=y*i
7     return y # Retourne n! Retour de la fonction (optionnel)
```

Regardons plus précisément les différentes parties:

1. l'entête d'une fonction commence par « `def` » suivi du nom de la fonction (*camelCase*), suivi des paramètres formels mais sans préciser leurs types.
2. ce n'est pas obligatoire mais c'est une bonne pratique de décrire la fonction en suivant le format `"" description ""`. Il est également possible d'informer le type de retour et les types des arguments de la manière suivante:

Type de l'argument
Type du retour
  
**def factorielle(n: int)-> int:**

cependant la précision des types n'impose pas le type des variables, il s'agit juste d'une information reprise dans l'aide de la fonction en faisant « `help(objet)` »:

```
>>> help(factorielle)
```

```
Help on function factorielle in module __main__:
```

<b>factorielle(n: int) -&gt; int</b> Calcule la factorielle de n
---

3. l'assertion est une condition de validité pour rentrer dans la fonction, elle permet de vérifier les hypothèses d'entrée. Une assertion se compose du mot « `assert` » suivi d'une condition à satisfaire et d'un message d'erreur qui sera joint à l'exception si la condition n'est pas vérifiée. L'utilisation d'une assertion n'est pas obligatoire mais est fortement utile pour débugger un programme, l'avantage c'est qu'il est toujours possible de désactiver les assertions lors du lancement du programme en rajoutant l'option « `-O` ».
4. vient ensuite le corps de la fonction. La fonction pourra définir de nouvelles variables locales en plus de celles créées via ses arguments formels. La fonction pourra aussi utiliser des variables externes à la fonction à condition qu'elles existent déjà mais ce n'est pas une pratique conseillée. Il est également possible en Python de définir une variable globale au sein même de la fonction: « `global nom` », cependant cette façon de faire risque de rendre votre programme complexe à comprendre.

```
#Variable globale au module
cpt = 0
def cube(a):
    global cpt
    cpt = cpt + 1 #Incrément le cpt (indique le nbr d'appels)
    return a * a * a
```

A chaque appel de la fonction « `cube` », la variable « `cpt` » sera incrémentée de un, ce qui permettra de savoir le nombre de fois qu'elle a été exécutée.

5. le retour de la fonction si retour il y a. Dans ce cas, l'instruction « `return` » permet de retourner une ou plusieurs variables sous forme d'un tuple.

Regardons maintenant les différentes façons offertes par le langage Python pour définir et préciser les arguments d'une fonction. D'abord il faut savoir que Python permet d'avoir des valeurs par défaut pour ses paramètres formels et permet lors de l'appel de référencer les paramètres formels à partir de leurs noms (paramètres nommés).

Un paramètre effectif peut être de 3 sortes:

- un paramètre **positionnel**, c'est la position du paramètre qui déterminera le paramètre formel associé, c'est la technique la plus courante.

- un paramètre **nommé**, on précisera à l'appel le nom du paramètre et sa valeur.
- un paramètre **par défaut**, dans ce cas il ne sera pas précisé lors de l'appel.

### Définition d'arguments en précisant des valeurs par défaut:

Pour définir une valeur par défaut à un paramètre formel, il suffit de le faire suivre par « `=` » et sa valeur par défaut. Il est tout à fait possible d'avoir plusieurs paramètres avec des valeurs par défaut mais ceux-ci doivent **toujours** se mettre après les paramètres positionnels, càd les paramètres sans valeur par défaut. Aucune confusion ne doit être possible au moment de l'appel.

**y sera à 2 si on ne précise pas ce paramètre à l'appel**

```
def power(x: float,y: int =2)->float:
```

Ainsi l'exécution de `power(5.0)`, calculera  $5^2$  comme le montre le test unitaire suivant:

```
self.assertEqual(mymath.power(5.0),25)
```

### Définition d'arguments effectifs à partir des noms des arguments formels:

Chaque argument formel possède un nom, Python autorise à l'appel de la fonction de préciser la valeur des paramètres via leurs noms. Ainsi la fonction « `power` » pourrait également être appelée des façons suivantes:

---

#### Algorithme 11.5 Arguments nommés en Python

---

```
def power(x: float,y: int = 2)->float:
    ...#corps de la fonction
#Exemples d'appels
power(x = 2, y = 3)
power(y = 3, x = 2) #l'ordre ne doit pas être respecté
power(x = 2) #car y possède une valeur par défaut
power(2 , y = 3) #possibilité de combiner des arguments positionnels suivis
    → d'arguments nommés
```

---

#### Remarques:

- l'ordre des paramètres ne doit pas nécessairement être respecté
- si on précise à l'appel des paramètres positionnels, ceux-ci doivent être placés en premier lieu
- les noms précisés à l'appel de la fonction doivent correspondre aux noms des paramètres formels

### Définition d'un nombre variable d'arguments:

Comme en Java, Python possède une syntaxe pour autoriser un nombre variable d'arguments. Ceux-ci ne doivent pas être obligatoirement du même type comme c'est le cas en Java. Pour définir un paramètre autorisant un nombre variable d'arguments, il faut précédé le nom du paramètre par une « `*` »:

---

**Algorithme 11.6** Nombre variable d'arguments en Python

---

```
#Le premier paramètre est un varArgs
def affiche( *chaine , separateur = ',' ):
    for c in chaine:
        print(f' {c} ',end=separateur)
    print()

#Ici le paramètre "separateur" prendra sa valeur par défaut
affiche(1,2,'T',4,5,'_')
#Ici on précise le séparateur avec un paramètre nommé
affiche(1,2,'T',4,5,separateur='_')
```

---

Voici le résultat affiché:

1 , 2 , T , 4 , 5 , _ ,	← Le séparateur est ',', celui par défaut
1 _ 2 _ T _ 4 _ 5 _	← Le séparateur est '_', celui précisé à l'appel

---

**Définition d'un nombre variable d'arguments nommés:**

Nous venons de voir que Python permet d'envoyer un nombre variable d'arguments qui seront stockés dans une liste mais Python permet aussi d'envoyer un nombre variable d'arguments nommés, ceux-ci seront stockés dans un dictionnaire. Un dictionnaire est une structure (clé→valeur), qui sera accessible via un paramètre formel précédé par deux astérisques « `**` ». Regardons cela sur un exemple:

---

**Algorithme 11.7** Nombre variable d'arguments nommés en Python

---

```
#Envoi de plusieurs arguments nommés
def connectionDb( **param ):
    for elem in param.items():
        print(f'clé: {elem[0]} ==> valeur: {elem[1]}')

#Exemple d'appel
connectionDb(login='admin',password='adminPW')
...
```

---

Voici le résultat affiché:

```
clé: login ==> valeur: admin
clé: password ==> valeur: adminPW
```

---



# Chapitre 12

## Les entrées sorties

Pour interagir avec un programme, vous avez généralement besoin de lui fournir des données et il doit aussi pouvoir vous fournir des résultats. Par défaut l'ordinateur utilise la console comme entrée et sortie standard. Mais de manière plus générale, on peut lire et écrire des données en provenance d'un fichier. Il est également possible d'établir une communication entre plusieurs processus en utilisant différents protocoles mais ceci sort du cadre de ce cours.

La notion de fichier va être liée au système de fichiers de votre système d'exploitation, mais ces aspects ne seront que très peu abordés ici.

Quand on parle de fichier, on va généralement distinguer un fichier texte d'un fichier binaire mais en réalité il n'y a pas tellement de différence car un fichier texte contient les codes binaires des caractères (liés à l'encodage utilisé). Lorsqu'on ouvre un fichier texte comme « *readme.txt* », votre système d'exploitation va utiliser l'encodage par défaut de votre système d'exploitation pour convertir les codes binaires en caractères, mais vous pourrez aussi l'afficher en binaire (hexadécimal) dans la plupart des lecteurs de fichier:

Voici le contenu d'un fichier texte codé avec l'encodage « WIN-1252 » :

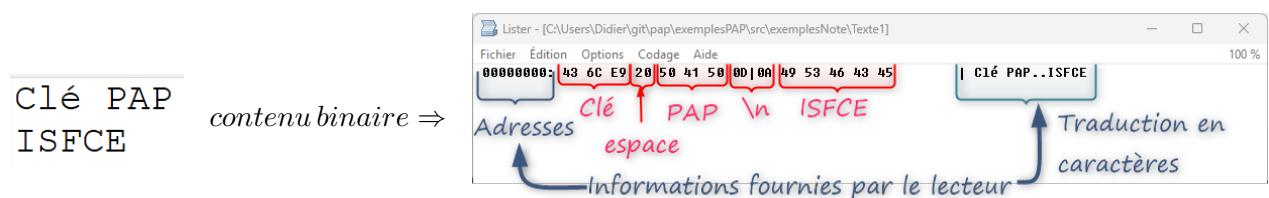


IMAGE 12.1 – Fichier texte et son codage binaire en WIN-1252

Voici le même fichier texte mais codé en UTF8, on peut constater que le « é » nécessite 2 bytes dans cet encodage:



IMAGE 12.2 – Fichier texte et son codage binaire en UTF8

Comme vous pouvez le constater sur les exemples ci-dessus, il est important de connaître l'encodage du fichier pour pouvoir le lire correctement. Pour une écriture, il faudra également savoir quel encodage

on désire utiliser pour le fichier de destination. Maintenant, je vous rassure, dans la plupart des cas on utilise l'encodage standard de votre ordinateur (système d'exploitation) et il sera utilisé par défaut.

## 12.1 Lecture et écriture sur la console

Les lectures et les écritures se font généralement sur des fichiers mais pour avoir une interaction basique entre un programme et un utilisateur, on utilise la console pour lire des données entrées par l'utilisateur et afficher les données produites par le programme. La console représente une sorte de fichier spécial, on parle d'entrée et de sortie standard.

En pseudo-code, l'entrée et la sortie s'exprime très simplement:

---

### Algorithme 12.1 Lecture et écriture standard

---

```
read txt
write txt
```

---

#### 12.1.1 Lecture sur la console:

La console se comporte comme un fichier texte, ainsi une lecture revient à lire une suite de caractères. La clôture de cette ligne se fera suite à l'appui de la touche de validation « ENTER ».

Si la variable de destination n'est pas de type « chaîne de caractères », il faudra convertir le texte vers le type de cette variable, c'est ce qu'on appelle **parser** un texte. *Parser* un texte peut provoquer une exception si le texte n'est pas convertible vers le type de destination: "P12HT" n'est pas convertible vers un entier et provoquera une exception!

Python possède l'instruction « *input* » pour lire un texte sur la console tout en permettant d'afficher une message exprimant l'entrée attendue. Le code suivant va demander à l'utilisateur d'entrer son nom et ensuite son age. Le nom est un texte et ne posera pas de difficulté par contre l'âge devra être converti en entier à l'aide d'un type-casting « *int( texte )* »:

---

```
nom=input("Entrez votre nom: ")
python age = int(input("Entrez votre age: "))
print(f'{nom} vous avez {age} ans')
                                         Entrez votre nom: Lola
                                         ⇒ Entrez votre age: 25
                                         Lola vous avez 25 ans
```

---

Derrière l'instruction « *input* », on retrouve une écriture sur la sortie standard « *stdout* » suivi d'une lecture sur l'entrée standard « *stdin* », voici un exemple de code qui utilise les entrées/sorties standards pour demander votre nom et ensuite l'afficher:

---

```
import sys
python sys.stdout.write("Entrez votre nom: ")
nom = sys.stdin.readline()
print(f'Bonjour {nom}')
                                         Entrez votre nom: Lola
                                         ⇒ Bonjour Lola
```

---



#### Remarque sur les exceptions en Python

Python n'impose pas d'envelopper l'ouverture d'une ressource dans un bloc « *try/except/finally* ». Cependant lorsqu'on utilisera un fichier à la place de la console, la gestion des exceptions sera plus que conseillée pour éviter le plantage de l'application! En effet, si par exemple un fichier n'existe pas, on peut demander à l'utilisateur de corriger le problème sans nécessairement devoir arrêter le programme. On verra que *Java* impose l'utilisation d'un bloc « *try catch* ».

En Java la syntaxe est plus complexe, on va utiliser ici une technique classique mais il existe aussi des classes comme « `Scanner` » qui facilitent la tâche pour lire plusieurs données ayant des types différents.

L'entrée standard « `System.in` » représente un « `InputStream` », il faut l'envelopper dans un objet « `InputStreamReader` », ce dernier peut aussi être enveloppé dans un « `BufferedReader` » pour optimiser les lectures. D'autre part, l'ouverture d'une ressource doit être entouré d'un bloc « `try catch` » pour traiter les éventuelles exceptions. Vous constaterez sur l'exemple ci-dessous qui la ressource « `br` » est déclarée au sein du bloc « `try (br...) catch` », ainsi la ressource sera automatiquement fermée à la sortie du bloc quelle que soit la situation.

Java ne permettant pas d'afficher un message lors de la lecture comme l'instruction « `input` » de Python, ainsi, il faudra précéder la lecture par une écriture du message désiré.

Sinon malgré un code initial plus complexe, la lecture elle-même s'effectue en une seule ligne!

---

### Algorithme 12.2 Lecture sur la console avec un `InputStream` et un `BufferedReader`

---



```
try (var br = new BufferedReader(new
    InputStreamReader(System.in))) {
    System.out.print("Entrez votre nom: ");
    String nom= br.readLine();
    System.out.print("Entrez votre age: ");      Entrez votre nom: Lola
    int age= Integer.parseInt(br.readLine());    ⇒ Entrez votre age: 25
    System.out.println(nom+ " vous avez "+age+" "  Lola vous avez 25 ans
        + ans");
} catch (IOException e)
    {System.err.println("erreur");}
}
```

---

L'instruction « `Integer.parseInt(string)` » permet de convertir le texte en entier, vous trouverez des méthodes similaires avec « `Double.parseDouble(string)` », « `Boolean.parseBoolean(string)` »,...

Voici une autre technique plus simple qui consiste à utiliser la classe `Scanner`:

---

### Algorithme 12.3 Utilisation de la classe Scanner

---



```
Scanner scan = new Scanner(System.in);
System.out.print("Entrez la taille de v: ");
int n = scan.nextInt();
System.out.print("Entrez votre nom: ");
String nom = scan.next();
int[] v = new int[n];
System.out.printf("%s Entrez les %2d
    entiers:\n",nom,n);
for (int i = 0; i < n; i++)
    v[i] = scan.nextInt();
System.out.print("Voici le vecteur: ");
for (int e : v)System.out.print(e+" ");
scan.close();
```

Entrez la taille de v: 2
 Entrez votre nom: Lola
 Lola Entrez les 2 entiers:
 ⇒ 6
 8
 Voici le vecteur: 6 8

---

#### 12.1.2 Écriture sur la console

l'écriture sur la console est un processus plus simple, on peut cependant regarder certains points plus techniques:

- ☛ afficher un code spécial

☛ afficher un caractère à partir de son code Unicode

☛ formater un texte lors de l'affichage

## Les codes spéciaux

Chaque caractère possède un code binaire associé mais certains codes ne correspondent pas à des caractères, ceux-ci ont des utilités spécifiques comme une tabulation, un passage à la ligne, un retour chariot,... On peut par contre les exprimer à l'aide d'un escape, comme par exemple « `\c` » représente un *carriage return* (retour chariot, référence aux anciennes machines à écrire) et a comme code hexadécimal « `0xD` » et « `\n` » représente un passage à la ligne et a comme code hexadécimal « `0xA` ».

Voici un tableau qui reprend les principaux escape-codes ainsi que leurs représentations en décimal et en hexadécimal:

Escape	Fonction	Code décimal	Code hexadécimal
<code>\n</code>	<i>new Line</i>	10	0xA
<code>\r</code>	<i>carriage return</i>	13	0xD
<code>\t</code>	<i>tab</i>	9	0x9
<code>\\\</code>	\	92	0x5c
<code>\b</code>	<i>backspace</i>	8	0x8
<code>\'</code>	<i>apostrophe</i>	39	0x27
<code>\"</code>	<i>guillemet</i>	34	0x22

TABLEAU 12.1 – Principaux *escape codes* valables en Java et en Python

## Afficher un caractère à partir de son code Unicode

Dans la plupart des langages, les caractères sont maintenant exprimés en Unicode 16 bits, il est possible d'afficher un caractère à partir de son code Unicode en utilisant un escape suivi du code hexadécimal:

```
>>> print('\u00E3F')    >>> print('\u20AC')
ß                      €
```

## Formatage du texte lors de l'affichage

Python possède une nouvelle syntaxe pour formater une chaîne de caractères avec les « *f'tring* » alors que java possède un « `printf` », regardons cela sur un exemple:

---

**Algorithme 12.4** Affichage d'un texte formaté

---



```
nom = "Lola"
age = 25
taille = 1.57
print(f'{nom} a {age:2} ans et {taille:4.3} mètre')
```

Lola a 25 ans et 1.57 mètres

Codes de formatage en Python:

<https://docs.python.org/3/library/stdtypes.html#old-string-formatting>



```
String nom="Lola"; int age=25;float taille=1.57f;
System.out.printf("%s a %2d ans et %1.3f mètres", nom , age ,
taille);
```

Lola a 25 ans et 1,570 mètres

Codes de formatage en Java:

<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Formatter.html#syntax>

---

## 12.2 Lecture et écriture sur des fichiers

Nous allons nous intéresser dans cette partie à la lecture d'un fichier et à l'écriture sur un fichier.

Avant tout, il faut savoir quel type de fichier on veut manipuler, s'agit-il d'un fichier binaire ou d'un fichier texte ? On va commencer par lire des fichiers contenant du texte.

Quand on manipule un fichier on doit suivre les étapes suivantes:

1. ouverture du fichier en précisant: type d'ouverture (lecture/écriture), type de fichier (texte, binaire), type d'encodage pour un fichier texte
2. lecture/écriture des données, le plus souvent le fichier sera parcouru séquentiellement du début à la fin mais il est généralement aussi possible de se positionner à des endroits dans un fichier.
3. fermeture du fichier. Des données risquent d'être perdues si on oublie de le fermer!
4. traitement des erreurs

Les opérations sur un fichier peuvent entraîner des exceptions pour signaler par exemple que le fichier n'existe pas ou que la lecture ou l'écriture ne peuvent pas se réaliser.

## 12.3 Lecture d'un fichier texte

Pour lire des données sur un fichier texte, il faut connaître les informations suivantes:

- ☛ l'endroit où se situe le fichier et bien entendu son nom
- ☛ l'encodage du fichier (l'encodage par défaut sera celui de votre système d'exploitation)
- ☛ comment est structuré le fichier pour pouvoir le lire convenablement
- ☛ comment se spécifie le passage à la ligne!
  - en Windows, le passage à la ligne sera codé par 2 codes: « 0D » et « 0A »
  - en Linux, le passage à la ligne sera codé par 1 code: « 0A » (passage à la ligne)

### 12.3.1 Spécifier le fichier et son chemin

La plupart du temps, il est possible de spécifier le nom du fichier et surtout son chemin de différentes façons. La structure d'un chemin sera aussi dépendante du système d'exploitation.

- ☛ Nom du fichier:

- il faut connaître son nom et son extension

- ☛ Chemin d'un fichier:

- attention au caractère séparateur des répertoires, il faudra le spécifier d'une façon différente en fonction du langage et du système d'exploitation:

```
#Exemples de chemins Windows séparateur '\'
"c:\user\vo\" #en java car '\' est un caractère d'escape
"c:/user/vo/"    #format souvent accepté
#Exemple en Linux. Sensible à la casse!
"/home/utilisateur/"
```

- chemin absolu ou relatif, il est aussi dépendant du système d'exploitation et du langage:
  - un chemin absolu démarre de la racine du répertoire
  - un chemin relatif peut être relatif à l'endroit courant ou relatif par rapport à la zone de l'utilisateur,...

### Spécifier le nom et le chemin d'un fichier Java

- ☛ Pour un chemin absolu c'est très simple:

```
String nom="C:\\\\Users\\\\Didier\\\\monFichier.txt";
File file = new File(nom);
```

- ☛ Pour un chemin relatif:

« .. / » permet de remonter vers le répertoire parent

« . / » désigne le répertoire courant

- rechercher un fichier dans un « package »: le fichier sera dans le dossier du code compilé:

```
String nom="monFichier.txt";
//pour avoir un objet URL
URL url = MaClasse.class.getResource(nom);
File file = url.getFile();
//ou pour avoir un objet InputStream
InputStream stream = MaClasse.class.getResourceAsStream(nom);
```

- chemin relatif au sein d'un répertoire:

```
String nom=".\\data\\monFichier.txt";
File file = new File(nom)
```

### Spécifier le nom et le chemin d'un fichier Python

Comme pour Java, Python possèdent plusieurs modules reprenant un ensemble de fonctions pour gérer le système de fichiers. Dans notre cas, on s'intéresse plus particulièrement à accéder à un fichier, cependant les conventions pour définir un chemin sont différentes d'un système d'exploitation à l'autre (Windows, Linux, Mac,...), regardons donc succinctement ce que Python propose:

Rappelez-vous d'abord ces quelques principales différences concernant les chemins:

	Windows	Linux
entête d'un chemin absolu	un lecteur comme « c:\ »	il n'y qu'une racine: « / »
séparateur	« \ »	« / »
sensible à la casse	non	oui

Voici quelques modules intéressants:

Modules	Remarques
module « os »	accéder au système de fichiers
module « pathlib »	permet de gérer des chemins indépendamment du système d'exploitation
...	
Exemples:	

```
import pathlib
nom = pathlib.PureWindowsPath('c:\Program Files', 'Thonny', 'data.txt')
#donnera: c:\Program Files\Thonny\data.txt
nom = pathlib.PurePosixPath('/user', 'Didier', 'data.txt')
#donnera: /user/Didier/data.txt
```

### 12.3.2 Lire les données d'un fichier texte ligne par ligne

Si on regarde la structure d'un fichier *CSV* (Comma Separated Values), il est constitué de lignes où chaque élément est séparé par un caractère particulier, la première ligne du fichier peut aussi contenir des entêtes de colonnes:

Prenons comme exemple le fichier suivant « achats.txt » codé en UTF8 qui commence par des entêtes de colonnes où le « ; » est utilisé comme séparateur:

```
Prénom;Nom;Montant
Alain;Chabat;12.70
Lola;Cool;175.25
Didier;VO;679.99
```

Il est possible de traiter ce fichier de pleins de façons différentes mais j'ai opté pour une lecture ligne par ligne pour générer une liste d'objets « Achat » qui sera ensuite affichée à la console.

Dans la version Python, je crée une classe « Achat » pour mémoriser de manière plus propre chaque ligne du fichier.

Voici quelques spécificités de ce code:

- ☛ le fichier est ouvert en lecture avec un encodage « utf8 »
- ☛ lors de la lecture, je retire le « \n » en fin de ligne avec « rstrip('\n') »
- ☛ le « split(';) » permet de générer un liste avec les éléments contenus dans la ligne et séparés par un « ; »
- ☛ « try except » permet de traiter les erreurs d'ouverture du fichier

---

**Algorithme 12.5** Lecture d'un fichier texte ligne par ligne

---

#Définition d'une classe Achat

```
class Achat:
    def __init__(self, prenom, nom, montant):
        self.prenom = prenom
        self.nom = nom
        self.montant = montant

try:
    #Ouverture d'un fichier en mode lecture, codé en UTF8
    f = open('c:/exemples/achats.txt', 'r', encoding='utf8')
    achats = [] #liste vide des achats
    #Affiche les entêtes de ligne
    for txt in f.readline().rstrip('\n').split(';'):
        print(f'{txt:10}', end=' ')
    print('\n-----')
    for l in f: #pour chaque ligne du fichier
        l = l.rstrip('\n') #supprime le passage à la ligne
        txt = l.split(';') #split en une liste
        achat = Achat(txt[0], txt[1], float(txt[2])) #Crée un achat
        achats.append(achat) #ajout à la liste
    #affiche les achats
    for achat in achats:
        print(f'{achat.prenom:10}{achat.nom:10}{achat.montant:7.2f}')
    f.close #fermeture du fichier
except FileNotFoundError as e:
    print('ERREUR: ', e)
```

---

Prénom	Nom	Montant
<hr/>		
Alain	Chabat	12.70
Lola	Cool	175.25
Didier	VO	679.99

---

Dans la version Java, j'utilise un « Record » qui est idéal dans ce cas.

Voici quelques remarques sur ce code:

- ☛ le « Scanner » est ouvert dans un bloc « try catch », de ce fait il sera fermé automatiquement.
  
  
  
- ☛ scan.nextLine() charge la ligne suivante
  
  
  
- ☛ split(";") permet de générer un vecteur avec chaque élément de la ligne séparé par un « ; »

---

**Algorithme 12.6** Lecture d'un fichier texte ligne par ligne

---

```

public class LectureCSV {
    //Record pour définir un achat
    public record Achat(String prenom, String nom, double montant) {
    };

    public static void main(String[] args) {
        File file = new File("c:/exemples/achats.txt");
        //liste d'achat
        List<Achat> liste=new ArrayList<>();
        try (Scanner scan = new Scanner(file,"utf8")) {
            // lecture des entêtes de colonnes:
            if (scan.hasNext()) {
                for (String txt:scan.nextLine().split(";"))
                    System.out.printf("%1$-10s",txt);
                System.out.println("\n-----");
            }
            while (scan.hasNext()) {
                // charge une ligne, splittée en mots
                String[] ligne = scan.nextLine().split(";");
                liste.add(new Achat(ligne[0], //charge le prénom
                                    ligne[1], //charge le nom
                                    Double.parseDouble(ligne[2])//charge le montant
                ));
            }
            //affiche la liste
            //"%1$-10s" indique le 1er paramètre est un string à aligner à gauche sur 10 caractères.
            liste.forEach(a->{System.out.printf("%1$-10s%2$-10s%3$-7.2f",a.prenom,a.nom,a.montant);
                System.out.println();});
        } catch (FileNotFoundException e) {
            System.err.println("Fichier introuvable: " + e.getLocalizedMessage());
        }
    }
}

```

---

Prénom	Nom	Montant
<hr/>		
Alain	Chabat	12,70
Lola	Cool	175,25
Didier	VO	679,99

---

## 12.4 Écriture dans un fichier texte

Pour écrire dans un fichier texte, il faut d'abord savoir si l'on doit créer un nouveau fichier, remplacer un fichier existant, ou rajouter des données à la fin d'un fichier. Ensuite il faut ouvrir le fichier en écriture et spécifier l'encodage à utiliser.

Reprenons la liste d'achats de l'exemple précédent et ajoutant la à un fichier « listeAchats.txt ».

Dans l'exemple, le fichier sera créé s'il n'existe pas encore, sinon les données seront rajoutées à la suite du fichier.

### 12.4.1 Écriture dans un fichier en Python

La commande « open » avec ses paramètres permet de préciser plusieurs informations dont voici les principales

1. le nom du fichier (obligatoire)
2. le mode d'ouverture (lecture/écriture) ==> défaut= 'read'

mode	signification
r	ouverture en lecture
w	ouverture en écriture
x	Écriture sur un nouveau fichier. Le fichier ne peut pas exister!
+	ouverture en lecture écriture

3. le type de fichier (binaire/texte) ==> défaut = 'texte'

code qui se combine avec le mode d'ouverture	signification
r <b>b</b> , w <b>b</b>	fichier binaire
r <b>t</b> , w <b>t</b>	texte (défaut)

4. l'encodage ==> défaut (celui du système)

Comme vous le savez un fichier doit toujours être fermé pour éviter la perte de données, pour ce faire Python possède l'instruction « **with** » qui se charge de fermer le fichier pour vous.

L'exemple suivant va sauvez dans un fichier une liste d'achats au format *CSV*, sans définir les entêtes de ligne.

Le fichier sera ouvert en mode « ajout », s'il n'existe pas encore, il sera créé.

---

**Algorithme 12.7** Écriture dans un fichier texte (Python)

```

import sys
#Définition d'une classe Achat
class Achat:
    def __init__(self,prenom,nom,montant):
        self.prenom = prenom
        self.nom = nom
        self.montant = montant

#Crée une liste d'achats
achats=[Achat("Didier","VO",124.90),Achat("Lola","COOL",123.77)]
#Ouverture du fichier avec l'encodage
nom = 'c:/exemples/listeAchats.txt'
try:#gère les exceptions d'erreurs
    #ouvre le fichier en append avec un encodage 'utf8'
    with open(nom,'a',encoding = 'utf8') as f:
        for achat in achats:
            #écrit une ligne avec le séparateur ';' et une fin de ligne
            f.write(achat.prenom+';'+achat.nom+str(achat.montant)+'\n')
except PermissionError as e:#gestion d'erreur de permission
    #Sortie sur l'output d'erreur standard
    sys.stderr.write(f'Le fichier existe déjà: {e}')

```

Voici le contenu du fichier après une exécution:

```

Didier;VO;124.9
Lola;COOL;123.77

```

Voici ce qui se passe si on écrit dans un dossier sans avoir les droits:

```

Le fichier existe déjà: [Errno 13] Permission denied: 'c:/listeAchats2.txt'

```

---

### 12.4.2 Écriture dans un fichier en Java

Dans le cas d'une écriture, on utilise souvent un *buffer* dans lequel les données seront préalablement écrites, il ne faudra donc pas oublier de fermer le fichier pour vider le *buffer*.

J'ai utilisé l'objet « *FileWriter* » pour écrire dans un fichier, il permet de préciser via ses paramètres:

**1<sup>er</sup> paramètre** le nom du fichier

**2<sup>ème</sup> paramètre** l'encodage

**3<sup>ème</sup> paramètre** le mode append « *true* »

Le ressource est ouverte dans le bloc « *try* », ainsi elle sera fermée automatiquement.

---

**Algorithme 12.8** Écriture dans un fichier texte (Java)

```

public class EcritureCSV {
    //Record pour définir un achat
    public record Achat(String prenom, String nom, double montant) {
    };
    public static void ecrireCSV() {
        //crée une liste d'achat
        List<Achat> achats = new ArrayList<>();
        achats.add(new Achat("Didier", "VO", 124.90));
        achats.add(new Achat("Lola", "COOL", 123.77));

        //Rajoute les achats à la fin d'un fichier
        //Le fichier sera automatiquement fermé ici!
        try (var out = new BufferedWriter("//utilise un buffer pour des raisons de performance
                                         //fichier, encodage utf8 , true ==> ajout au fichier
                                         new FileWriter("c:/exemples/listeAchats.txt", Charset.forName("utf8"),true));) {
            for (var achat : achats) //écriture sous format CSV avec comme séparateur le ";"
                out.append(achat.prenom + ";" +
                           + achat.nom + ";" +
                           + Double.toString(achat.montant) + "\n");
        } catch (IOException e) {
            System.err.println("Oups erreur: "+e);
        }
    }
}

```

Voici le contenu du fichier après une exécution:

```
Didier;VO;124.9
Lola;COOL;123.77
```

Voici le message d'erreur lorsque je n'ai pas les droits sur le dossier:

```
Oups erreur: java.io.FileNotFoundException: c:\listeAchats.txt (Accès refusé)
```

---

# Chapitre 13

## La gestion des exceptions

Lorsqu'un bloc de code déclenche une exception, en principe celle-ci va arrêter le programme. Cependant vous avez, pour certains types d'exception, la possibilité de prendre le contrôle lorsqu'une exception se produit et ainsi la gérer sans devoir arrêter le programme. Prenons un exemple classique d'ouverture d'un fichier, il est possible par exemple que ce fichier n'existe pas à l'endroit précisé ou que le nom du fichier est incorrect, il est donc possible via le traitement de l'exception d'avertir l'utilisateur pour qu'il puisse corriger sa faute et ceci sans devoir relancer le programme.



# Chapitre 14

## Les notions de classe et d'objet

Malgré le fait que le cours de PAP ne comprend pas dans son programme les notions de programmation orientée objet, il est cependant difficile de l'éviter complètement, par exemple en Python tout est objet, en Java tout est objet en dehors des types primitifs! Et d'ailleurs, il n'est pas question d'éviter ces aspects, cependant nous n'aborderons que les notions élémentaires de la programmation orientée objet.

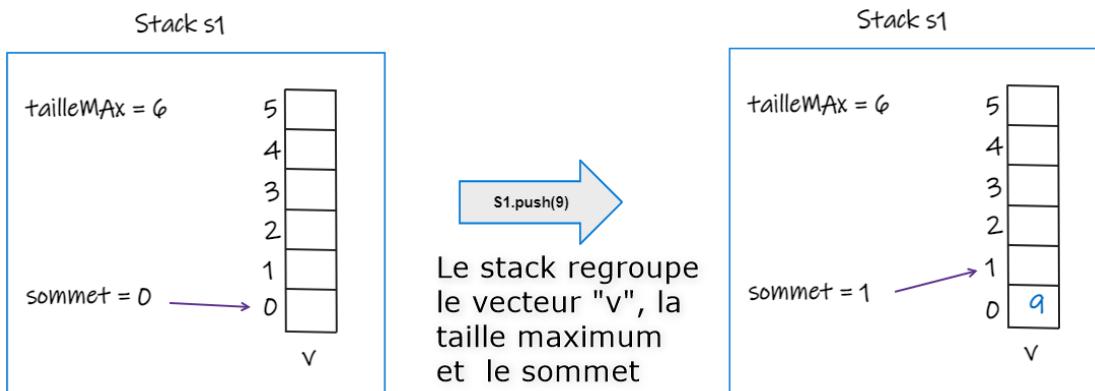
Prenons l'exemple d'un *stack*(15.2) que nous allons implémenter, on peut d'une part identifier les différentes méthodes (fonctions) qui s'appliquent à un *stack* et d'autre part les données nécessaires à stocker ses éléments:

- ☛ méthodes principales d'un *stack*:
  - mettre un élément sur le *stack*: « `push (T elem)` »
  - retirer l'élément au sommet du *stack*: « `T pop()` »
  - obtenir l'élément au sommet du *stack* sans le retirer: « `T top()` »
  - savoir s'il est vide: « `boolean empty()` »
- ☛ attributs pour gérer le *stack* (*hyp.*: on utilise un vecteur pour l'implémenter)
  - taille maximum du *stack* (à définir lors de sa création): « `int tailleMax` »
  - un vecteur d'éléments de taille « `tailleMax` »: « `T[] v` »
  - un entier qui désigne le sommet du *stack*, le premier emplacement libre: « `sommet` »

On constate que ces variables sont fortement liées, elles forment un tout! Il est donc logique de les regrouper au sein d'une même structure, dans notre cas: un objet.

D'un autre coté, l'utilisateur ne va pas directement utiliser ces variables, elles devront être manipulées uniquement par les méthodes de l'objet.

Voici une représentation visuelle du stack et son évolution lorsqu'on lui rajoute un élément:



En programmation orienté objet, les données que l'on appelle les **attributs** ont généralement une visibilité privée, cela signifie qu'il n'est pas possible d'avoir accès à ces variables en dehors de l'objet.

Les méthodes de l'objet (procédures et fonctions) ont par contre directement accès à ces attributs généralement via un mot réservé qui désigne l'objet: « `self` » en Python et « `this` » en Java.

### Algorithme 14.1 Accès aux attributs à partir des méthodes d'un objet

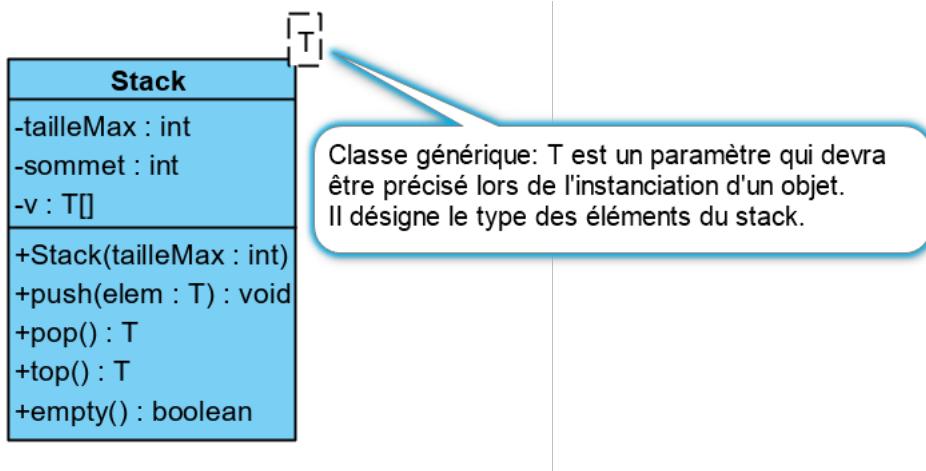
```
 python self.__sommet = self.__sommet + 1
 Java
this.sommet++;
//ou directement sans utiliser this, s'il n'y a pas de conflit avec
→ une variable locale de même nom:
sommet++;
```

## 14.1 La notion de classe et la construction d'un objet

Les langages orientés objets, possèdent une structure appelée « `Class` » qui permet de regrouper les données et les méthodes d'un objet ainsi que les attributs de classe (*static*) et les méthodes de classe (*static*).

Jusqu'à présent tous les attributs et les méthodes étaient des méthodes statiques, càd des éléments accessibles directement sur la classe sans devoir créer d'objet!

Voici une représentation de la classe Stack en utilisant le langage de modélisation UML:



Les attributs d'objets dépendent de l'objet, ainsi si l'on crée 3 piles à partir de la classe `Stack`, chaque objet aura ses propres attributs (`v`, `sommet` et `tailleMax`). Regardons maintenant comment construire un objet.

### CODE

Si on utilise une liste à la place d'un vecteur, la taille maximum n'est plus nécessaire. On pourrait éventuellement spécifier la taille initiale de la liste. Une liste peut s'agrandir automatiquement lorsqu'on lui rajoute des éléments mais n'oublions pas que ceci a un coût!

### 14.1.1 Construction d'un objet

Un objet est une **instance** d'une classe, on parle d'**instancier** un objet, c'est le terme technique synonyme de « créer un objet ».

## Le constructeur

Pour instancier un objet, il faut appeler une méthode particulière que l'on appelle le **constructeur**, il est possible d'avoir plusieurs constructeurs pour permettre de construire un objet de différentes façons.

En *UML* le constructeur porte le même nom que la classe, ce qui est aussi le cas dans plusieurs langages de programmation, en Python par contre le nom d'un constructeur s'appelle « `__init__` ».

Un constructeur est une méthode qui ne possède pas de paramètre de retour car il retourne par défaut l'objet créé. Le code du constructeur permettra d'initialiser les attributs de l'objet.

---

### Algorithme 14.2 Accès aux attributs à partir des méthodes d'un objet

---

```
class Stack:
    def __init__(self,tailleMax):
        self.sommet = 0
        self.tailleMax = tailleMax
        self.v = tailleMax*[0] #initialise une liste avec tailleMax
                           ↳ éléments
```

---

```
Class Stack<T>{
    public Stack(int tailleMax){
        this.tailleMax = tailleMax;
        this.sommet = 0;
        this.v = (T[]) new Object[tailleMax];
    }
```

---

## Création d'un objet

Linstanciation d'un objet nécessite de faire appel à son constructeur ou à un de ses constructeurs. Dans notre exemple le *stack* est un *stack* générique dans le sens où le type de ses éléments est représenté par « *T* », il faudra précisé ce paramètre lors de linstanciation. En Python par contre, le type des éléments du *stack* ne doit pas être précisé car il peut être quelconque, ainsi le type générique ne concerne pas ce langage.

---

### Algorithme 14.3 Instanciation d'un *stack*

---

```
#Crée un stack avec une taille initiale de 10 cases
s1 = Stack(10)
s1.push(9)
print(s1.pop())

//Crée un stack d'entiers de taille maximum à 10
Stack<Integer> s1=new Stack<>(10);
s1.push(9);
System.out.println(s1.pop());
```

---

## 14.2 Visibilité des méthodes et des attributs

Un objet peut être vu comme une boîte noire à laquelle on accède via une interface représentée par les méthodes publiques de l'objet. Par défaut, les attributs de l'objet sont privés et donc invisibles en dehors des méthodes de l'objet. On parle en terme technique d'**encapsulation**, ce qui signifie que l'on cache le « comment » et on rend visible uniquement les services applicables sur l'objet, ce qu'on peut faire sur l'objet.

Il existe plusieurs visibilités mais dans notre cas, nous nous intéresserons qu'aux visibilités privée et publique:

Visibilité	UML	Java	Python
publique	+	public	
privée	-	private	____ (préfixe des attributs et méthodes par 2 soulignés)

TABLEAU 14.1 – Visibilité et ses représentations

Un constructeur a souvent une visibilité publique mais il est tout à fait possible d'avoir des constructeurs avec une visibilité privée. Dans ce cas, la classe possède une ou plusieurs méthodes statiques (de classe) qui se chargeront de créer les objets en appelant un constructeur privé. Cette façon de procéder à plusieurs intérêts mais ces aspects ne seront pas abordés dans ce cours.

## 14.3 Destruction des objets

Un objet créé devra être détruit à un moment donné. Il existe 2 techniques différentes:

- ☛ destruction manuelle des objets

chaque objet créé devra être détruit manuellement en appelant un [destructeur](#), une méthode spécifique qui devra libérer les ressources de l'objet et libérer l'espace mémoire occupé par celui-ci. C'est le cas des langages de programmation comme *C++*, *Delphi*, ...

- ☛ utilisation d'un « [garbage collector](#) »

lorsqu'un objet n'est plus accessible, plus aucune référence ne pointe vers l'objet, il sera détruit automatiquement par un processus qui s'appelle le « *garbage collector* », c'est le ramasseur d'ordures. Ce mécanisme est bien pratique mais il a aussi ses inconvénients que nous n'aborderons pas ici. Java et Python utilisent cette technique. Si « *s1* » désigne un *stack*, en faisant « *s1 = null* » en Java ou « *s1 = None* » en Python et qu'aucune autre référence pointe vers le *stack*, celui-ci sera détruit à un certain moment quand le ramasseur d'ordure se mettra au travail. En Python, on peut aussi assigner n'importe quelle autre objet à la variable sans devoir nécessairement mettre « *None* », en Java aussi sauf que l'on ne peut pas mettre autre chose qu'un objet Stack ou null.

## 14.4 Implémentation

Pour illustrer la création d'une classe et linstanciation d'objets, on va reprendre l'exemple du *stack* que l'on va coder en *Java* et en *Python*. Bien entendu, il existe plusieurs façons de réaliser un *stack*, par exemple en exploitant une classe existante dans les bibliothèques du langage, mais ici notre but premier est de montrer le concept d'objet et non pas directement d'utiliser des classes existantes. D'ailleurs la plupart des classes existantes, ne correspondent pas précisément au contrat d'un *stack*, souvent ce sont des classes qui possèdent plusieurs autres méthodes que l'on ne devrait pas pouvoir appeler sur un *stack*.

Si on crée sa propre classe pour implémenter un *stack*, il reste le choix interne qui consiste à opter pour un vecteur, une liste, une liste chaînée,... Vu l'exemple présenté depuis le début de ce chapitre, je vais utiliser un simple vecteur pour coder le *stack* en Java et utiliser une liste en Python puisque le langage ne propose pas de vecteur à proprement parler.

### 14.4.1 implémentation du *stack* en Java

Comme expliqué précédemment, cette implémentation du *stack* utilisera un vecteur. Comme on aimerait bien pouvoir créer un *stack* pour stocker des objets différents comme un *stack* d'entiers ou un *stack* de personnes; on va définir une classe générique où le type des éléments sera précisé lors de l'instanciation de la classe. Pour permettre ce mécanisme, il faudra que notre *stack* soit un *stack* d'objets et non pas un *stack* de type primitif.

```
public class Stack<T> { //Le stack générique où T est le paramètre
    private final int tailleMax; //taille maximum du stack
    private int sommet; //désigne la première case libre
    private T[] v; // le vecteur d'éléments de type T

    /**
     * Constructeur
     * @param tailleMax
     */
    public Stack(int tailleMax) {
        this.tailleMax = tailleMax;
        //création d'un vecteur de taille tailleMax. Code un peu technique
        //→ car le type est générique. Ici on crée un vecteur d'Object que
        //→ l'on caste en vecteur de type T
        v = (T[]) new Object[tailleMax];
    }

    /**
     * Ajoute un élément sur le stack assertion
     *
     * @assertion sommet < tailleMax
     * @param elem élément à rajouter
     */
    public void push(T elem) {
        assert sommet < tailleMax : "Stack Overflow";
        v[sommet] = elem;
        sommet++;
    }

    /**
     * retire l'élément du sommet du stack assertion
     *
     * @assertion: sommet>0
     * @return l'élément du sommet du stack
     */
    public T pop() {
        assert sommet > 0 : "Stack Underflow";
        sommet--;
        return v[sommet];
    }

    /**
     * Renvoit l'élément au sommet du stack
     */
}
```

```

*
 * @assertion: sommet>0
 * @return l'élément au sommet du stack
 */
public T top() {
    assert sommet > 0 : "Stack Underflow";
    return v[sommet - 1];
}

public boolean empty() {
    return sommet == 0;
}
}

```

Regardons maintenant comment instancier un *stack* :

```

//Crée un stack de taille maximum à 10
Stack<Integer> s1=new Stack<>(10);
s1.push(9);
//Crée un stack d'étudiants avec maximum 40 entrées
Stack<Etudiant> s2= new Stack<>(40);
...
//suppression des stacks
s1 = null;
s2 = null;

```

#### 14.4.2 Implémentation du stack en Python

En Python le problème est légèrement simplifié car le type des éléments ne doit pas être spécifié et qu'il n'existe pas de "simple" vecteur. Ainsi, on utilisera une liste. On constatera que l'implémentation en Python ci-dessous, utilisera directement les fonctionnalités d'une liste tout en limitant celles-ci aux besoins du *stack*. On pourrait aussi envisager un code qui crée une liste avec une taille initiale, ce qui permettrait une amélioration de performance mais le code serait un peu plus long.

La programmation orientée objet en Python nécessite quelques points d'attention:

- ☛ le constructeur doit s'appeler `__init__` et doit posséder le paramètre « `self` ».
- ☛ les attributs peuvent être déclarés (ou rajoutés) à l'objet dans n'importe quelle méthode via `self`: « `self.cpt = 0` » rajoute l'attribut public « `cpt` ».
- ☛ les attributs d'objet sont accessibles via « `self.attribut` » ou « `self.__attribut` » pour un attribut privé
- ☛ les attributs privés sont préfixés par 2 soulignés
- ☛ chaque méthode d'objet doit spécifier « `self` » comme premier paramètre

---

**Algorithme 14.4** Implémentation du *stack* en Python
 

---

```
class Stack:

    def __init__(self):
        self.__v=[]

    def push(self,elem):
        self.__v.append(elem)

    def pop(self):
        assert len(self.__v) > 0 ; "Stack Underflow"
        elem=self.__v[-1]
        del(self.__v[-1])
        return elem

    def top(self):
        assert len(self.__v) > 0 ; "Stack Underflow"
        return self.__v[-1]

    def empty(self):
        return len(self.__v)==0
```

---

Regardons maintenant comment instancier un *stack* :

```
//Crée un stack
s1 = Stack();
s1.push(9);
//suppression du stack
s1 = None;# ou associer un autre objet
```

## 14.5 Les classes immuables

Une classe immuable va permettre de créer des objets non modifiables. Les objets immuables ont beaucoup d'avantages dans le cadre du développement informatique car un tel objet peut être utilisé, copié sans risque, puisqu'il ne changera pas. Le problème quand un objet peut être modifié, c'est qu'il faut s'assurer que l'objet ne soit pas utilisé lors de sa modification et que toutes ses copies soient mises à jour. Les objets immuables auront plusieurs avantages dont voici les principaux:

- ☛ possibilité de les charger en cache sans risque
- ☛ ils sont thread-safe, plusieurs processus peuvent les utiliser en même temps
- ☛ permet de faire des économies de mémoire
- ☛ ce sont des clés idéales pour les dictionnaires, les ensembles
- ☛ ...

### 14.5.1 Crédation d'une classe immuable

Une classe immuable doit être une classe

- ☛ que l'on ne peut pas étendre (final)
- ☛ tous ses attributs doivent être non modifiable (final)
- ☛ si un attribut est de type objet, il faut s'assurer de ne jamais rendre cet objet accessible et si nécessaire de renvoyer une copie de celui-ci.
- ☛ la référence à l'objet ne doit jamais être renvoyée, ici on parle de « this » en Java ou « self » en Python.

### 14.5.2 Les structures immuables en Java et en Python

Voici les principales classes qui génèrent des objets immuables. Il est également possible en *Java* de créer ses propres classes pour que leurs objets soient immuables (cf.: « final »).

Java	Classes immuables	Python	Classes immuables
booléens	Boolean	booléens	bool
entier	Byte, Short, Integer, Long, BigInteger, BigDecimal	entier	int
réels	Float, Double	réels	float
caractères	Character, String	caractères	str
structure	Record	structure	tuple
divers	Color,...		frozensets

TABLEAU 14.2 – Classes immuables en Java/Python

### 14.5.3 Les records Java

Java 17 a introduit la notion de Record, il s'agit d'une écriture synthétique pour créer une classe immuable<sup>1</sup>. Les objets créés à partir d'un Record auront automatiquement les getters, les méthodes « *toString* », « *Equals* », « *HashCode* ». Cette structure est donc très intéressante dans plusieurs cas par exemple on pourrait implémenter un *Stack* à partir d'une liste chaînée d'éléments:

Appelons un élément de la chaîne un maillon, dans le cadre d'un *stack*, chaque maillon stockera l'information à mémoriser plus l'adresse du maillon suivant.

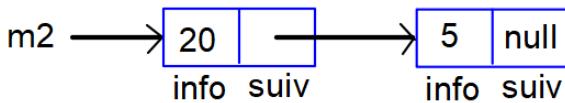
Ne connaissant pas le type des éléments à stocker sur le *stack*, le Maillon aura donc un type générique « *K* »:

```
record Maillon<K>(K info, Maillon<K> suiv)
```

Si je créé 2 maillons contenant des entiers que je relie, cela donnera ceci:

```
Maillon<Integer> m1= new Maillon<>(5,null);
Maillon<Integer> m2= new Maillon<>(20,m1);
```

Voici sa représentation graphique:



1. Classe immuable: en réalité ce n'est pas la classe qui est immuable mais bien les objets créés à partir de celle-ci!

# Chapitre 15

## Des structures de mémorisation courantes

En dehors des vecteurs que nous avons déjà étudiés, il existe des structures plus élaborées comme des listes chaînées, des piles, des files,...

Par exemple, les listes Java sont des objets qui utilisent comme implémentation, soit un vecteur soit une liste chaînée.

Dans l'exemple suivant, j'ai déclaré deux listes, la 1<sup>ère</sup> utilise une *ArrayList* qui cache un vecteur alors que la 2<sup>ème</sup> utilise comme implémentation une *LinkedList* qui cache une liste chaînée:

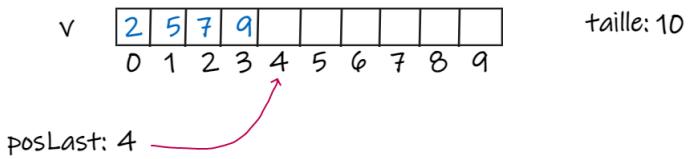
```
List<Integer> liste1 = new ArrayList<>(); //utilise un vecteur  
List<Integer> liste2 = new LinkedList<>(); // utilise une liste chainée
```

Parmi les termes énoncés, il ne faut pas confondre la liste que l'on retrouve en Python ou en Java avec une liste chaînée. La liste Python ou Java permettent d'offrir des services comme rajouter un élément, supprimer un élément, accéder à une élément via un indice, insérer un élément,... et tout ceci sans devoir se préoccuper de la taille mémoire à réserver. C'est l'implémentation de l'objet qui gère le tout pour vous. Cependant pour réaliser ces services, il faut mémoriser les éléments soit à l'aide d'un vecteur ou d'une liste d'éléments chaînés.

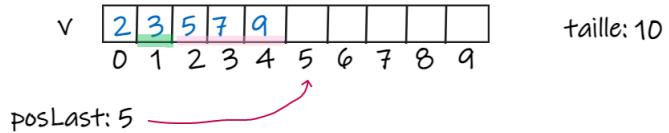
Regardons maintenant plus précisément le fonctionnement d'une liste chaînée et ses avantages et inconvénients.

### 15.1 Liste simplement chaînée

Prenons un vecteur contenant des éléments et supposons que l'on désire intercaler un élément à une position dans le vecteur, cela nécessiterait de décaler plusieurs éléments vers la droite, du moins s'il reste de la place dans le vecteur car un vecteur à une taille définie à sa création. Pour gérer une structure de mémorisation à partir d'un vecteur, on utilisera généralement un vecteur de taille n et une variable entière qui indique la position du dernier élément (ou la première place de libre):

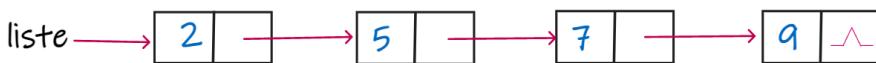


Insertion d'un élément 3 en position 1 ==>



⭐ Les éléments 9, 7 et 5 ont du être déplacés d'une position

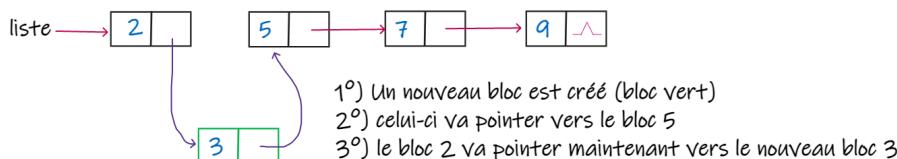
Dans le cadre d'une liste simplement liée, chaque élément est mémorisé dans une structure qui contient l'information plus un lien vers le bloc suivant:



Bien entendu chaque élément consomme un peu plus de mémoire car il faut mémoriser l'élément plus un pointeur vers le bloc suivant. Par contre, la liste ne comprend que ses éléments (plus pointeurs), pour l'agrandir, il suffit de chaîner des nouveaux blocs, ce n'est pas comme un vecteur qui a une taille fixe définie à sa création.

L'image suivante montre le comportement lorsque l'on insère un nouvel élément à l'intérieur de la chaîne:

Ajout d'un élément 3 entre le 2 et le 5



On peut ainsi rajouter, tant que la mémoire le permet, autant de blocs que nécessaire et ceci sans devoir faire de décalages.

Par contre un gros désavantage est qu'il n'est plus possible d'avoir accès à un bloc via son indice (sa position), pour accéder à la 4<sup>ème</sup> case, il faut parcourir la liste de sa tête de liste en passant par les éléments 2, 3 et 5.

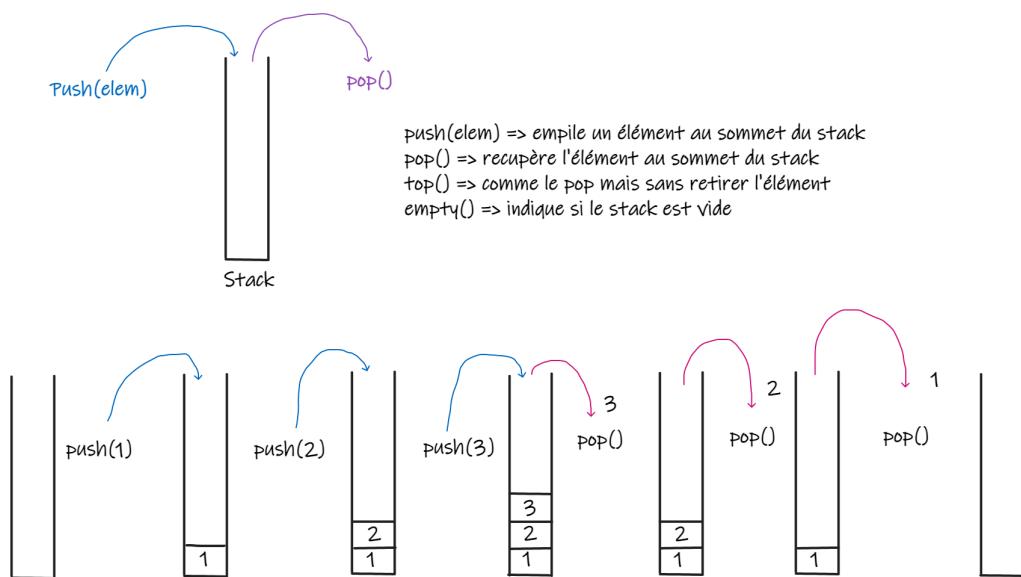
Avantages et inconvénients d'une liste par rapport à un vecteur:

- 👉 possibilité de rajouter autant d'éléments que nécessaire tant que la mémoire le permet
- 👉 pas besoin de définir une taille initiale, seuls les éléments utiles sont présents

- 👉 possibilité d'intercaler ou de supprimer des éléments dans la liste sans devoir faire de décalages
- 👉 consomme un peu plus de mémoire pour chaque bloc
- 👉 gestion plus complexe qu'un vecteur
- 👉 impossible d'y accéder via un indice, ce qui est un gros désavantage pour certains problèmes
- 👉 accès plus lent

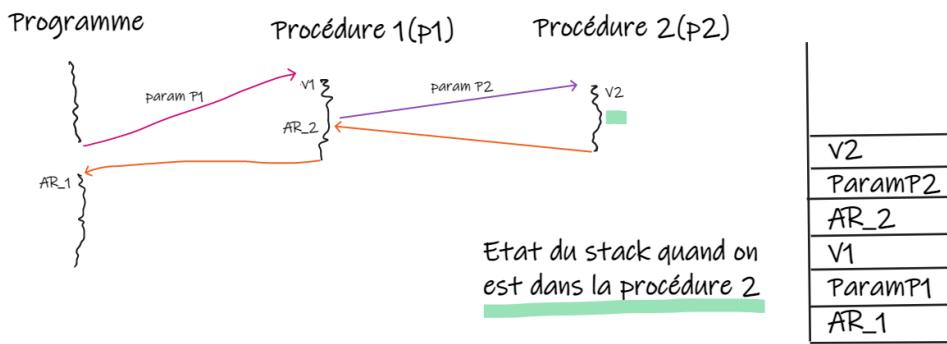
## 15.2 Le stack (la pile)

Le *stack* est une structure de mémorisation du type *LIFO* pour **Last In First Out** (Dernier Entré Premier Sorti), c'est une structure à empilement d'où le terme de « pile » en français. On peut représenter le *stack* de la manière suivante avec ses opérations classiques PUSH et POP :



Le *stack* est fortement utilisé comme structure de mémorisation, le processeur de votre ordinateur possède des instructions *push* et *pop* permettant de mémoriser des informations en mémoire, une utilisation du *stack* est de mémoriser les variables locales d'une procédure ainsi que l'adresse de retour:

Lorsqu'un programme appelle une procédure, il stocke sur le *stack* de l'application les paramètres à envoyer à la procédure ainsi que l'adresse de retour (l'adresse désignant l'endroit où il faut revenir quand la procédure se termine). Ensuite, la procédure y empile ses variables locales. Lorsqu'une procédure appelle une autre procédure, ce même mécanisme va se reproduire. Ce n'est que lorsqu'une procédure se terminera que les données propres à celle-ci seront retirées du *stack*.



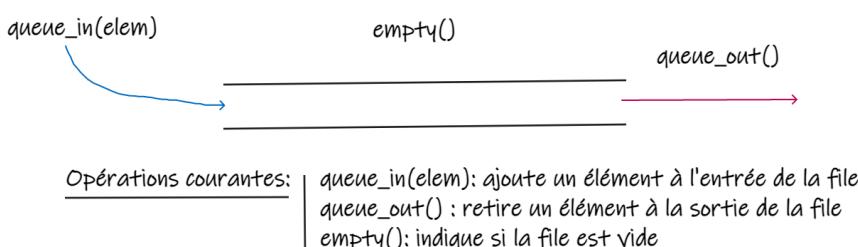
Si vous avez besoin d'un *stack* au sein de votre application, vous pourrez certainement en trouver dans les libraires fournies avec le langage de programmation mais si celles-ci ne vous conviennent pas, rien ne vous empêche de réaliser votre propre structure de *stack*. Dans ce cas, il faudrait créer une classe « *Stack* » dans laquelle vous implémenteriez un vecteur ou une liste chaînée pour stocker les éléments.

A votre avis, si vous deviez implémenter une « *file* », vous opteriez plutôt pour une liste chaînée ou un vecteur?

### 15.3 La queue (file)

Une autre structure courante de mémorisation est la file **FIFO (First In First Out)**, comme un file d'attente dans un magasin, on rentre d'un coté et on attend son tour pour sortie de l'autre coté. On l'utilise moins souvent que le *stack* mais on peut la retrouver par exemple pour gérer une file d'attente d'impression, une liste d'opérations en attente d'exécution. Par exemple les *frameworks* graphiques comme *Swing*, *JavaFX*, utilisent une file dans la quelle on injectera toutes les tâches graphiques à exécuter, ces tâches seront retirées au fur et à mesure de la file par un *thread* spécifique qui se chargera de les exécuter.

Les opérations classiques sur une file sont *queue\_in*, *queue\_out*, *empty* pour respectivement ajouter un élément à l'entrée de la file, retirer un élément en sortie de file et savoir si elle est vide:



On peut bien entendu aussi envisager une opération équivalente à *queue\_out* mais sans retirer l'élément de la file ( le pendant de *top* du *stack*) ou une opération « *length()* » qui indique le nombre d'éléments dans la file.

Comme pour le *stack*, il est généralement possible de trouver, dans les librairies de votre langage de programmation préféré, une structure pour gérer une file et si celle-ci ne vous convient pas, mettez

vos gants pour la créer vous-même. Comme pour le *stack*, l'implémentation d'une file nécessite d'avoir une liste simplement chaînée ou un vecteur.

A votre avis, si vous deviez implémenter une « file », vous opteriez plutôt pour une liste chaînée ou un vecteur ?



# Chapitre 16

## La récursivité

Un problème de taille  $n$  est considéré comme récursif s'il peut se résoudre par une nombre fini de problèmes similaires de taille inférieure à  $n$  et que nous connaissons la solution du problème pour certaines valeurs de  $n$ .

La plupart des langages de programmation permettent de réaliser des appels récursifs, ce qui signifie qu'une fonction peut s'appeler elle-même. Bien entendu, pour que le programme ne tourne pas sans fin, il faut avoir une condition d'arrêt!

Commençons par un problème mathématique simple qui peut s'exprimer de manière récursif:

### 16.1 Le calcul de la factorielle de $n$

Le calcul de la factorielle de  $n$  peut s'exprimer de manière récursive de la façon suivante:

$$\begin{aligned} n! &= n * (n - 1)! \text{ pour } n > 0 && \text{structure récursive} \\ 0! &= 1 && \text{condition d'arrêt} \end{aligned}$$

On constate son écriture récursive avec sa condition d'arrêt  $0! = 1$ . Il est donc possible de calculer la factorielle de  $n$  à partir d'un algorithme récursif:

---

#### Algorithme 16.1 Calcul de « $n!$ » en utilisant la récursivité

 python

```
def factorielle(n):
    if n==0: return 1 #condition d'arrêt
    return n * factorielle(n-1) #Appel récursif
```

---

On peut constater la simplicité de l'algorithme, cependant l'écriture récursive n'est pas intéressante ici car on peut très bien calculer  $n!$  en partant d'une problème de taille 0 vers un problème de taille  $n$ , comme le montre l'algorithme suivant:

---

#### Algorithme 16.2 Calcul de « $n!$ » sans récursivité

 python

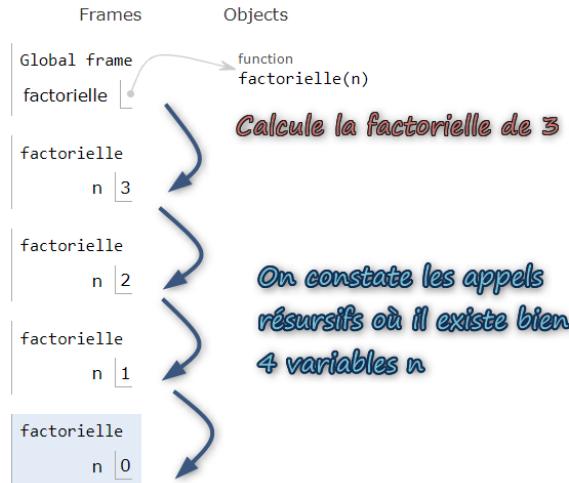
```
def factorielle(n):
    res=1
    for i in range(2,n+1):
        res = res * i
    return res # Retourne n!
```

---

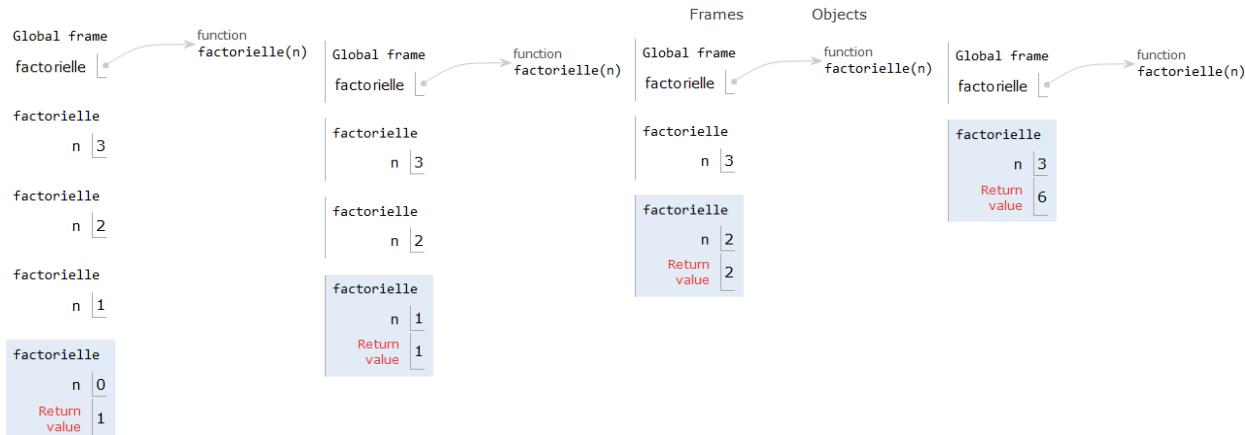
Regardons plus en détail le fonctionnement d'un appel récursif, rappelez-vous lors d'un appel de fonction, toutes les variables locales (paramètres d'entrée compris) seront créés sur le *stack* et au retour de la fonction ces mêmes variables seront supprimées! C'est ce comportement là qui permet de réaliser

des algorithmes récursifs! Regardez l'image suivante qui illustre l'état de l'application après 4 appels de la fonction « factorielle »:

1. dans le module « *main* », on appelle la fonction « *factorielle(3)* »
2. dans la fonction *factorielle* on appelle la fonction « *factorielle(2)* »
3. dans la fonction *factorielle* on appelle la fonction « *factorielle(1)* »
4. dans la fonction *factorielle* on appelle la fonction « *factorielle(0)* »



A ce stade, la fonction ne devra plus faire d'appel récursif car elle connaît la valeur de  $0!$ , le *stack* pourra se vider au fur et à mesure:



Constatations à partir de cet exemple:

- ☛ la récursivité utilise le *stack* système et le fait grandir à chaque appel récursif
- ☛ le code de l'algorithme est très synthétique et facile à comprendre, mais pas pour autant facile à créer! En effet, il faut d'abord découvrir l'aspect récursif d'un problème et ensuite pouvoir écrire son algorithme. La récursivité est souvent un casse tête pour les étudiants.
- ☛ le calcul de la factorielle peut se réaliser facilement sans faire appel à la récursivité et donc sans faire grandir le *stack*. Sa version récursive n'est donc pas intéressante en dehors de son aspect pédagogique!

Prenons un autre exemple récursif ou la récursivité ne sera pas intéressante alors que le problème est souvent exprimé de manière récursive. Mais rassurez-vous, la récursivité est très intéressante dans de nombreuses situations comme nous le verrons plus loin!

## 16.2 Calcul du $n^{\text{ème}}$ nombre de Fibonacci

Rappelez-vous de cette fameuse suite de Fibonacci où le  $n^{\text{ème}}$  nombre est le résultat de la somme des 2 nombres précédents:

$F_0$	$F_1$	$F_2$	$F_3$	$F_4$	$F_5$	...
0	1	1	2	3	5	...

Cette suite s'exprime naturellement de manière récursive:

$F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$	structure récursive
$F_1 = 1$	
$F_0 = 0$	conditions d'arrêt

Cependant l'écriture récursive ne sera pas performante du tout car d'une part, comme pour la factorielle, on peut partir d'un problème de taille 1 vers un problème de taille n et d'autre part la récursivité va calculer plusieurs fois la même chose:

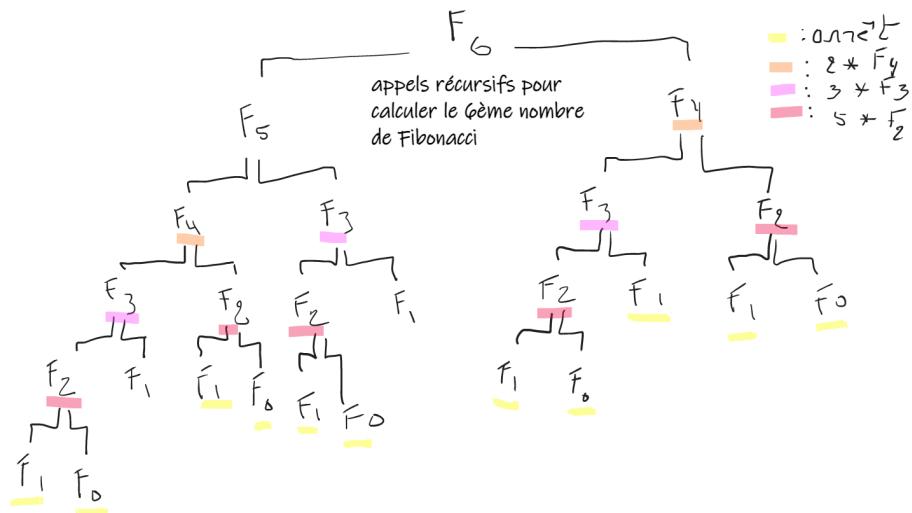


IMAGE 16.1 – Appels récursifs pour calculer le 6<sup>ème</sup> nombre de Fibonacci

On constate que pour calculer  $F_6$  l'algorithme récursif calculera 2 fois  $F_4$ , 3 fois  $F_3$  et 5 fois  $F_2$ . Voici cependant l'écriture récursive écrite en Python:

---

### Algorithme 16.3 Calcul du $n^{\text{ème}}$ nombre de Fibonacci de manière récursive

---

```
def fibo(n):
    python if n<2: return n
            return fibo(n-1) + fibo(n-2)
```

---

Conclusion sur cet exemple:

- ☛ ce n'est pas parce que l'écriture récursive est évidente que l'algorithme sera performant! Particulièrement dans ce cas présent, on calculera plusieurs fois la même chose!
- ☛ ce problème peut facilement s'écrire de manière non récursive et performante. Le fait de profiter du *stack* n'est pas du tout utile!

Après avoir montré 2 exemples par très intéressant au niveau des performances, regardons des problèmes où la récursivité est beaucoup plus intéressante!

## 16.3 La revisite de la fonction « power(x,y) »

Dans le cadre des premiers exercices, nous avons réalisé une fonction « power(x,y) » avec  $y$  entier positif et nous avons trouvé une amélioration de l'algorithme en constatant que :

Si « $y$ » est pair	$x^y = x^{y/2} * x^{y/2}$	$\Rightarrow$ on calcule $x^{y/2}$ que l'on multiplie par lui-même
Si « $y$ » est impair	$x^y = x^{(y-1)/2} * x^{(y-1)/2} * x$	$\Rightarrow$ ici on aura une multiplication par $x$ en plus

Par exemple pour calculer  $2^{100}$ , il faudrait effectuer  $2^{50} * 2^{50}$ , ce qui implique 50 multiplications au lieu de 99, le gain est intéressant mais si on regarde le problème de plus près, on pourrait appliquer le même raisonnement avec  $2^{50} = 2^{25} * 2^{25}$  et ainsi de suite!

Mais dans un algorithme récursif, il faut une condition d'arrêt, condition que l'on retrouve ici lorsque l'exposant  $y$  est égal à 0, 1 ou 2 :

si $y > 1$ et est pair	$power(x, y) = power(x, y/2)^2$
si $y > 1$ et impair	$power(x, y) = power(x, (y - 1)/2)^2 * x$
si $y = 1$	$power(x, y) = x$
si $y = 0$	$power(x, y) = 1$

Remarques:

- ☛ on peut bien entendu aussi éliminer d'office les cas avec  $x=0$  ou  $x=1$
- ☛ éliminer le cas de  $0^0$

Reprendons l'exemple précédent avec le calcul de  $2^{100}$  pour voir le gain en performance:

$2^{100}$	Nombre de multiplications
$2^{50} * 2^{50}$	1
$2^{25} * 2^{25}$	1
$2^{12} * 2^{12} * 2$	2
$2^6 * 2^6$	1
$2^3 * 2^3$	1
$2^1 * 2^1 * 2$	2
	8

Le nombre de multiplications était de 99 avec une simple boucle « pour », 50 avec la première amélioration et seulement 8 avec l'algorithme récursif.

Voici un pseudo-code de la version récursive suivi par des versions en Java et en Python:

---

### Algorithme 16.4 PseudoCode de $power(x,y)$ récursif

---

**Require:**  $y \in \mathbb{N}$  et  $y \geq 0$

```

1: function POWER( $x,y$ )
2:   if  $y = 0$  then
3:     return 1
4:   end if
5:   if  $y = 1$  or  $x = 0$  or  $x = 1$  then // élimine les cas particuliers
6:     return  $x$ 
7:   end if
8:    $z \leftarrow y//2$  //  $z$  reçoit la division entière de  $y$ 
9:    $res = POWER(x, z)$  // appel récursif
10:   $res \leftarrow res * res$  // fait le carré du résultat
11:  if  $y$  est impair then
12:     $res \leftarrow res * x$ 
13:  end if
14:  return  $res$  // retourne le résultat
15: end function

```

---

Conclusions sur cet exemple:

- ☛ l'algorithme récursif est ici très performant, bien entendu il utilise le *stack* de l'application mais il ne serait pas possible de résoudre ce problème en partant d'un problème de taille 1 vers un problème de taille n. Il est cependant possible de dérécursifier un algorithme mais cela nécessiterait de créer son propre *stack*!
- ☛ on retrouve plusieurs conditions d'arrêt ici car on peut facilement éliminer des cas  $1^n$  ou  $0^n$  ou encore  $x^1$  ou  $x^0$
- ☛ l'algorithme peut aussi être facilement adapté pour traiter le cas où  $y$  serait négatif
- ☛ en fonction des langages de programmation, on pourrait utiliser un décalage droit pour calculer  $y/2$  et un opération binaire *and* «  $(y \& 1) == 1$  » pour savoir si «  $y$  » est impaire

Voici maintenant les écritures en Java et Python

---

#### Algorithme 16.5 La fonction $\text{Power}(x,y)$ en version récursive

---



```
def power(x: float, y: int) -> float:
    '''Calcule  $x^y$  avec  $y$  entier positif (récuratif)'''
    assert isinstance(x, (int, float)) and isinstance(y, int) and
        y >= 0, 'Erreur x: réel, y: entier positif'
    if y == 0: return 1
    if y == 1 or x == 0 or x == 1:
        return x
    res = power(x, y // 2)
    res = res * res
    #si y est impaire alors res<-res*x
    if y % 2 != 0:
        res = res * x
    return res
```

---



```
public static double power(double x, int y) {
    assert y >= 0 : "Y doit être un entier positif";
    // Elimine les cas exposant 0
    if (y == 0)
        return 1.0;
    if (y == 1 || x == 0 || x == 1)
        return x;
    double res = power(x, y >> 1);
    res *= res;
    // Si le nombre est impaire, je multiplie par x
    if ((y & 1) == 1) // ou (y % 2 != 0)
        res *= x;
    return res;
}
```

---

## 16.4 Le tri « Quicksort »

Comme son nom l'indique, le tri *QuickSort* est un tri rapide qui est souvent utilisé dans les librairies des différents langages mais avec des versions légèrement adaptées. Ce tri nous intéresse particulièrement ici car c'est un problème récursif.

Le principe d'un passage de l'algorithme consiste à positionner un nombre du vecteur à sa bonne place (le pivot) tout en s'arrangeant pour que les nombres à gauche du pivot soient inférieurs au pivot et ceux à droite du pivot soient supérieurs ou égaux au pivot:

La première étape consiste à prendre un nombre celui à droite de la zone à trier que l'on nomme le pivot:

Tri du vecteur de l'indice 0 à l'indice 7 (choix du pivot à l'indice 7)								
État initial du vecteur								
11	2	4	20	12	10	3	6	
0	1	2	3	4	5	6	7	

Ensuite on va positionner le pivot à sa bonne place tout en déplaçant les nombres inférieurs au pivot sur la partie gauche du vecteur et ceux supérieurs ou égaux au pivot sur la partie droite du vecteur: (la façon de faire sera expliquée plus loin)

Positionne le pivot à sa place								
<pivot			position finale du pivot			>=pivot		
3	2	4	6			12	10	11
0	1	2		3		4	5	6

Maintenant il ne reste plus qu'à trier indépendamment le sous-vecteur de gauche et le sous-vecteur de droite et ceci de manière récursive!

Tri de 0 à 2			à sa place	Tri de 4 à 7			
3	2	4	6	12	10	11	20
0	1	2	3	4	5	6	7

La condition d'arrêt sera la situation où la taille du sous-vecteur sera égal à 1. On peut très bien s'arrêter avant, par exemple lorsque la taille est inférieure à 10, pour appliquer un autre algorithme que le *quicksort* sur cette partie du vecteur car l'algorithme du *quicksort* est très efficace sur des grands vecteurs et nettement moins sur des petits vecteurs, d'où l'intérêt de mixer plusieurs techniques de tri.

Ainsi pour trier un vecteur « v », on appellera la fonction « quicksort(v,0,n-1) » pour un vecteur de taille n:

- ☛ « v »: désignant le vecteur, comme il s'agit d'une adresse, seule l'adresse sera envoyée à la fonction dans une variable locale (pas tout le vecteur)
- ☛ 0: désigne l'indice de la première case du vecteur qui sera recopiée dans une variable locale de la fonction
- ☛ n-1: désigne le dernier indice du vecteur, celui-ci sera aussi recopié dans une variable locale de la fonction

Voici une version en pseudo-code de cet algorithme :

**Algorithme 16.6** PseudoCode de l'algorithme du tri *QuickSort*


---

```

1: procedure QUICKSORT( $v, i, j$ )
2:   if  $j - i > 0$  then // condition d'arrêt taille $\leq 1$ 
3:      $indicePivot \leftarrow CALCULPIVOT(v, i, j)$  // organise la partie de vecteur en positionnant le
   pivot et retourne son indice
4:     QUICKSORT( $v, i, indicePivot - 1$ ) // appel récursif pour la partie de gauche
5:     QUICKSORT( $v, indicePivot + 1, j$ ) // appel récursif pour la partie de droite
6:   end if
7: end procedure

8: function CALCULPIVOT( $v, i, j$ )
9:    $pivot \leftarrow v_j$  // l'élément à droite sera le pivot
10:   $a \leftarrow i$  // a désigne l'indice gauche
11:   $b \leftarrow j - 1$  // b désigne l'indice droit, avant le pivot
12:  while  $a \leq b$  do // tq qu'on n'a pas trouvé la position du pivot
    while  $a \leq b$  and  $v_b \geq pivot$  do // On garde à droite les éléments  $\geq$  au pivot
13:     $b \leftarrow b - 1$ 
14:  end while
15:  while  $v_a < pivot$  do // On garde à gauche les éléments  $<$  au pivot
16:     $a \leftarrow a + 1$ 
17:  end while
18:  if  $a < b$  then // si a<b c'est qu'on doit inverser v[a] avec v[b]
19:     $v_a \rightleftharpoons v_b$ 
20:  end if
21:  end while
22:   $v_a \rightleftharpoons v_j$  // a désigne la position du pivot, on échange donc v[j] avec v[a] pour mettre le pivot
   à sa place
23:  return  $a$  // retourne la position du pivot
24: end function

```

---



# Chapitre 17

## Valider une séquence de transitions



# Chapitre 18

## Annexes

### 18.1 Résumé du langage de pseudo-code

Dans le cadre du syllabus, j'ai utilisé une syntaxe de pseudo-code en anglais (dépendant du module `latex` utilisé) mais sur papier vous pouvez suivre une syntaxe en français même utiliser une phrase pour exprimer une opération comme « si `a` est impaire alors », ou « créer une matrice de `n` sur `m` initialisée à 0 »,...

Le pseudo-code vous permettra d'exprimer un problème d'une manière moins formelle qu'avec un langage de programmation et de manière plus synthétique. Une fois le pseudo-code créé, il ne suffira plus qu'à l'implémenter dans un langage de programmation, la réflexion algorithmique se retrouve dans le pseudo-code, lors de la traduction dans un langage, il n'est plus nécessaire de réfléchir sur le fonctionnement du problème mais comment il faut traduire ce pseudo-code avec les spécificités du langage.

Dans le tableau suivant, j'ai indiqué une suite d'instructions en pseudo-code mais celle-ci n'est pas exhaustive.

Comme dans un pseudo-code, on n'indique pas le type des variables, il est indispensable de préciser avant un pseudo-code les points suivants:

1. les hypothèses avant de démarrer l'algorithme. Celles-ci pourront par exemple être vérifiées via des assertions.
2. les paramètres d'entrée, il faudra préciser le type ainsi que la signification de chaque paramètre
3. les paramètres de sortie s'il y en a, en précisant pour chaque paramètre son type et sa signification
4. les principales variables avec pour chacune d'elle le type et sa signification

- 1°) affectation :  $a \leftarrow 4$
- 2°) échange :  $a \leftrightarrow b$
- 3°) division réelle :  $a / b$   
entière :  $a // b$
- 4°) modulo :  $a \text{ modulo } b$  ou  $a \% b$
- 5°) test égalité :  $a = b$
- 6°) test si : si  $a < b$  alors  
sinon  $i \leftarrow i + 1$   
fin  $j \leftarrow j + 1$   
fin  $a < b$  faire  
fin
- 7°) tant que : tant que  $a \geq b$
- 8°) répéter : répéter  
fin  
fin
- 9°) pour : jusqu'à ce que  $a \geq b$   
pour  $i : 0 \rightarrow m$  faire  
fin pour  
pour  $i : m \rightarrow 0$  par pas de -1 faire  
fin pour
- 10°) indice matrice :  $M_{ij}$
- 11°) taille matrice :  $m \cdot \text{taille}$  ou Taille de  $m$
- 12°) fonction : fonction  $f(a, b)$   
retourne  $a + b$   
fin fonction
- 13°) procédure : procédure  $p1(a, b)$   
fin procédure
- 14°) nouveau vecteur :  $v \leftarrow \text{nouveau vecteur taille } n$   
nouvelle matrice :  $m \leftarrow \text{nouvelle matrice de taille } m \times m$

## 18.2 Installer Python

Python est un langage de programmation interprété datant de 1991 et conçu par Guido van Rossum. Vous trouverez sur le site officiel de Python [Python.org](http://Python.org) toutes les informations pour l'installer et

de nombreuses documentations pour s'initier et se perfectionner au langage.

Dans le cadre de ce cours, l'objectif est d'apprendre à programmer et d'exprimer un algorithme dans un langage de programmation actuel, nous n'allons donc pas étudier le langage dans tous ses coutures mais dans ses éléments essentiels.

L'interpréteur Python est généralement déjà présent si vous avez Linux comme système d'exploitation et même parfois sous Windows. Pour faire des exercices, il vous faut un environnement de développement qu'on appelle communément une IDE, il en existe plusieurs mais pour commencer avec le langage, je vais utiliser l'IDE *Thonny* qui embarque avec lui l'interpréteur Python.

## 18.3 Installer Java

```
//associe le nom de la propriété à la cellule
colOperation.setCellValueFactory(cellData -> new
    SimpleStringProperty(cellData.getValue().getNom()));
colOperation.setCellFactory(param -> new TableCell<>() {
    //Le bouton delete
    final Button btDel = new Button();
    //image du bouton
    final ImageView iconDel = new ImageView(new Image("/view/icon/trash.png"));
    //Conteneur du bouton
    final Pane paneBt = new StackPane();
    //ajout de l'image au bouton
    btDel.setGraphic(iconDel);
    //padding
    paneBt.setPadding(new Insets(2));
    //ajout du bouton au conteneur
    paneBt.getChildren().add(btDel);
    /*en récupérant la valeur de la cellule (nom de la propriété)
    je tente de supprimer la propriété dans la BD
    */
    btDel.setOnAction(event -> {
        //Récupère la propriété dans la liste observable du tableau
        Optional<Propriete> oProp = obsPropriete.stream().filter(e
            -> e.getNom().equals(((Button)
            event.getSource()).getId())).findFirst();

        if (oProp.isPresent()) {
            try {//tente de supprimer la propriété ds BD
                facade.deletePropriete(oProp.get());
                //supprime dans la liste observable du tableau
                obsPropriete.remove(oProp.get());
            } catch (ComposantException e1) {
                //affiche une fenêtre en cas d'erreur
                ctrl.showErreur(e1.getMessage());
            }
        }
    });
});
```



# **Divers**



## Liste des algorithmes

3.1	Création d'une constante . . . . .	11
3.2	Exemple de type énuméré (pseudo-code) . . . . .	13
3.3	Création d'une constante . . . . .	14
3.4	Référent vers un objet . . . . .	15
3.5	Assignation d'une fonction en Python . . . . .	15
3.6	Assignation d'une fonction en Java . . . . .	15
4.1	Le saut inconditionnel de type GOTO à éviter . . . . .	18
4.2	Rupture de séquence: « if then » . . . . .	19
4.3	Rupture de séquence: « <i>if then else</i> » . . . . .	20
4.4	Rupture de séquence avec des « <i>if</i> » imbriqués . . . . .	21
4.5	Rupture de séquence avec des « <i>if</i> » imbriqués . . . . .	22
4.6	Version du « selon que » en Python . . . . .	23
4.7	Version1 du <i>switch</i> en Java . . . . .	24
4.8	Version2 du <i>switch</i> en Java . . . . .	24
4.9	Version3 du <i>switch</i> en Java . . . . .	24
4.10	La boucle « <i>while</i> » en Java . . . . .	26
4.11	La boucle « <i>do while</i> » en Java . . . . .	26
4.12	La boucle « <i>while</i> » en Python . . . . .	27
4.13	Calcul du <i>pgcd</i> en Python . . . . .	27
4.14	Calcul du <i>pgcd</i> en Java . . . . .	28
4.15	Calcul de n! en Java . . . . .	29
4.16	Boucle « pour » en Python . . . . .	30
4.17	Calcul de « n! » en Python . . . . .	30
4.18	Calcul de « n! » version2 . . . . .	31
4.19	Boucle « for each » en Java . . . . .	31
5.1	Type Casting . . . . .	34
5.2	Opérateur ternaire . . . . .	42
6.1	Liste Python vs Liste Java . . . . .	43
6.2	Afficher l'adresse du vecteur . . . . .	45
6.3	Taille d'un vecteur . . . . .	45
6.4	Parcours d'un vecteur via les indices . . . . .	46
6.5	Parcourir chaque élément d'un vecteur . . . . .	46
6.6	PseudoCode pour la recherche du plus grand élément d'un vecteur . . . . .	47
6.7	Plus grand élément d'un vecteur en Java . . . . .	47
6.8	PseudoCode: compte le nombre de fois qu'une valeur existe dans un vecteur . . . . .	47

6.9	Compte le nombre de fois qu'une valeur existe dans un vecteur en Java . . . . .	47
6.10	Taille des éléments d'une matrice . . . . .	50
6.11	Parcours d'une matrice via ses indices . . . . .	50
6.12	Parcourir chaque élément de la matrice . . . . .	51
6.13	Afficher une matrice ligne par ligne en Java . . . . .	51
6.14	PseudoCode qui vérifie si une matrice carrée est symétrique . . . . .	52
6.15	Vérifie si une matrice carrée est symétrique (en Java) . . . . .	52
7.1	Modification d'un élément immuable d'une liste . . . . .	60
7.2	Copie profonde d'une liste . . . . .	61
7.3	Copie partielle d'une liste . . . . .	62
7.4	Parcourir les éléments d'une liste . . . . .	64
7.5	Somme de 2 matrices en Python v1 . . . . .	66
7.6	Somme de 2 matrices en Python v2 . . . . .	66
7.7	Parcourir les éléments d'un tuple . . . . .	68
7.8	Parcourir les éléments d'un ensemble . . . . .	69
7.9	Création et initialisation d'un dictionnaire . . . . .	70
7.10	Parcourir les éléments d'un dictionnaire . . . . .	71
8.1	Les types énumérés utilisés pour définir une carte d'un jeu de cartes . . . . .	74
8.2	Création d'un enregistrement « Carte » . . . . .	74
8.3	Version Python d'une carte . . . . .	74
9.1	Interface et implémentation . . . . .	76
9.2	Parcourir une liste Java . . . . .	79
9.3	Parcourir les éléments d'un ensemble Java . . . . .	81
9.4	Obtenir les éléments d'un dictionnaire . . . . .	83
11.1	Exemple de fonction . . . . .	88
11.2	Exemple de procédure . . . . .	88
11.3	Envoi classique des arguments en Java . . . . .	95
11.4	Envoi d'un nombre variable d'arguments en Java . . . . .	96
11.5	Arguments nommés en Python . . . . .	98
11.6	Nombre variable d'arguments en Python . . . . .	99
11.7	Nombre variable d'arguments nommés en Python . . . . .	99
12.1	Lecture et écriture standard . . . . .	102
12.2	Lecture sur la console avec un <i>InputStream</i> et un <i>BufferedReader</i> . . . . .	103
12.3	Utilisation de la classe Scanner . . . . .	103
12.4	Affichage d'un texte formaté . . . . .	105
12.5	Lecture d'un fichier texte ligne par ligne . . . . .	108
12.6	Lecture d'un fichier texte ligne par ligne . . . . .	109
12.7	Écriture dans un fichier texte (Python) . . . . .	111
12.8	Écriture dans un fichier texte (Java) . . . . .	112
14.1	Accès aux attributs à partir des méthodes d'un objet . . . . .	116
14.2	Accès aux attributs à partir des méthodes d'un objet . . . . .	117
14.3	Instanciation d'un <i>stack</i> . . . . .	117
14.4	Implémentation du <i>stack</i> en Python . . . . .	121
16.1	Calcul de « n! » en utilisant la récursivité . . . . .	129
16.2	Calcul de « n! » sans récursivité . . . . .	129

16.3 Calcul du $n^{\text{ème}}$ nombre de Fibonacci de manière récursive . . . . .	131
16.4 PseudoCode de $\text{power}(x,y)$ récursif . . . . .	132
16.5 La fonction $\text{Power}(x,y)$ en version récursive . . . . .	133
16.6 PseudoCode de l'algorithme du tri <i>QuickSort</i> . . . . .	135
18.1 Version 1 de la classe <i>Bic</i> . . . . .	149



## Liste des symboles

CamelCase	Le camel case est une notation consistant à écrire un ensemble de mots en les liant sans espace ni ponctuation, et en mettant en capitale la première lettre de chaque mot. La première lettre du premier mot peut être en bas ou haut de casse, selon la convention "Wikipedia"
HEAP	Le Heap ou la Tas en français désigne une structure de mémoire organisée comme un parking sans emplacement de taille fixe. Des blocs de mémoire y seront attribués en fonction de la demande et une fois libérées ils seront rajoutés à une FreeList qui contiendra les espaces disponibles. Le tas ressemblera à un morceau de fromage avec plein de trous.
IDE	integrated development environment
Octet	Un octet représente un groupe de 8 bits. Un bit étant un chiffre en base 2.
STACK	Un stack ou une pile en français désigne une structure de mémorisation d'empilement type LIFO (Last In First Out) ou dernier entré, premier sorti. Les fonctions standards d'un stack sont PUSH, POP, TOP et EMPTY
UML	Unified Modeling Language

IMAGE 18.1 – La classe Bic avec ses attributs privés et ses méthodes

---

### Algorithme 18.1 Version 1 de la classe *Bic*

---



Visibilité	Symboles en UML	Description
Attribut		Attribut visible uniquement dans la classe où il est déclaré.
Méthode		Méthode accessible depuis l'extérieur de la classe.
Méthode		Méthode accessible depuis la classe et ses sous-classes.

TABLEAU 18.1 – Visibilités

 Truc

 Remarque

Statique

Représentation en UML

TABLEAU 18.2 – attribut et méthode statique

//TODO

## Bibliographie

- [1] La tour de Hanoï
- [2] Le jeu de casse briques