Chapitre 1

Eléments de base

Ce chapître s'adresse surtout à un public qui n'a jamais programmé en JAVA . Il contient un bon nombre d'inexactitudes pour permettre une compréhension assez basique du langage et nous seront plus précis dans les chapîtres qui suivent.

1 Ecrire du JAVA

1.1 Où écrire ? Extension `.java`

Le langage JAVA est un langage orienté objet. La plupart du temps, on manipulera des objets en leur faisant faire des opérations. Un *classe* definit ce qu'un objet peut faire. Ceci deviendra plus clair dans le chapître 4, mais l'important pour le moment est qu'on va écrire du code JAVA s'écrit dans un fichier texte qui possède l'extension `.java`. Par convention, le nom d'une classe est une chaîne de lettres (sans espace) qui commence toujours par une lettre *majuscule*, par exemple `MaPremiereClasse`. Cette classe sera écrite dans le fichier `MaPremiereClasse.java`.

Note: Fichiers

On a indiqué que `MaPremiereClasse.java` est un fichier dit *fichier texte*. De façon abstraite, un fichier est un espace mémoire du disque dur composé de `0` et de `1` qui encodent de l'information. Selon le type de fichier, on utilisera différentes méthodes pour décoder le fichier. Un fichier texte est un fichier dont le contenu représente seulement une suite de caractères (i.e., la suite de `0` et `1` encode une suite de caractères qui peuvent être lus et compris par un humain). Par exemple, les fichiers `.doc` ou `.xls` ne sont pas des fichiers textes, ils encodent un contenu plus riche qui sera décodé par les applications `Word` ou `Excel`. Les fichiers `.html` sont des fichiers textes mais qui seront affichés facilement par un navigateur internet.

1.2 Instructions et commentaires

JAVA est un *langage*, il possède donc une grammaire pour écrire des phrases dont la syntaxe est correcte. La grammaire de JAVA est assez simple et nous allons l'introduire de façon informelle dans la suite. Pour que le code soit compréhensible par la machine, on utilisera un programme appelé *compilateur* qui va traduire le code JAVA que vous avez écrit en un code compréhensible par la machine. Cette étape de traduction est appelée *compilation*, nous en parlerons succinctement plus tard. Lors de la compilation la première étape est toujours de vérifier si la grammaire du langage est respectée. Lors de la seconde étape, le compilateur vérifie si toutes les instructions ont un sens, par exemple si une variable utilisée a bien été déclarée, si une fonction est appelée avec le bon nombre d'arguments et les bons types, etc.

Dans un code en JAVA , on aura deux grands types de « phrase » :

— soit on a une instruction en JAVA qui doit suivre la grammaire du langage et toujours se terminer par un point-virgule ; qui correspond au point à la fin d'une phrase [1].

— soit un commentaire, i.e. un texte qui va aider à expliquer le code (par exemple, il peut jouer le rôle de titre pour une partie de code ou expliquer le but du code qui suit). Un commentaire peut décrire le code, ou bien justifier les instructions (i.e. donner un argument qui prouve que le code est correct).

Il y a deux grandes manière d'écrire des commentaires :

— commentaire sur le reste de la ligne :

```
1 // la suite est un commentaire
```

— commentaire sur plusieurs ligne : il commence par `/*` et se termine par `*/`.

```
1 /* ceci est un commentaire
2  sur plusieurs
3  lignes */
```

— il exite ensuite un petit langage spécial qui sert à générer automatiquement de la documentation (des pages html) à l'aide de JAVADOC. Dans ce cas, le commentaire commence par `/**`, le commentaire lui même doit suivre un format particulier et le commentaire se termine par `*/`.

1.3 Un premier programme et affichage dans la console

Le grand classique : faire afficher la chaîne `Hello World` dans la console.

```
1 package introjava ;
2
3 // The classic!
4
5  public class HelloWorld {
6    public static void main(String[] args){
7      System.out.println(’’ Hello World!’’) ;
8    }
9 }
```

Nous allons décrypter ce code de manière grossière, nous verrons en détail chaque point un peu plus tard.

La première ligne indique que la classe fait partie du `package` appelé `introjava`. Pour le moment, on indiquera simplement qu'un package permet de rassembler un ensemble de classes. Cela permet entre autre d'avoir plusieurs classes de même nom dans plusieurs `packages` différents. Le nom complet de notre classe est en fait `introjava.HelloWorld`.

En ligne 3, on peut donc voir un commentaire.

`main` est une *méthode*, c'est à dire une fonction déclarée à l'intérieur d'une classe. La ligne 3 est la déclaration de la méthode : on va y trouver son nom, si elle retourne quelque chose ou non, ses arguments. Le code exécuté par l'appel de la méthode, ce que l'on appelera le *corps* de la méthode, se trouve entre les deux accolades. Ici le corps de la méthode se limite à une seule instruction ligne 7.
`main` est une méthode particulière, c'est la première méthode appelée quand un programme est exécuté.

1. On verra aussi que la définition de classes ou de fonctions ne se terminent pas par un point virgule ;

```
~$ javac introjava/HelloWorld.java
~$ java introjava/HelloWorld
Hello World!
~$
```

FIGURE 1.1 – Compilation et Excécution en ligne de commande

Le mot clé `static` indique que la méthode n'opère sur aucun objet (au moment de l'appel de `main`, il n'y a pas encore d'objet). La méthode ne retourne rien, ce qui est indiqué par le mot clé `void`. Finalement, on indique la visibilité de la méthode : ici, elle est `public`, ce qui veut dire que tout autre objet pourra utiliser cette méthode. On verra qu'on pourra définir la visibilité de plusieurs éléments en JAVA .
Entre parenthèse après le nom de la méthode on trouve la liste des arguments (les paramètres d'entrée de la méthode). Ici, il n'y a qu'un seul paramètre : il a pour nom `args` et c'est un tableau contenant des `String` (on le verra plus tard `[]` signifie un tableau, et ce qui précède `[]` est le type des éléments du tableau, ici il s'agit du type `String` qui représente une chaîne de caractères).
Le corps de la méthode est donc une seule instruction : cette instruction est d'afficher un message dans l'objet `System.out`, un objet qui représente la « sortie standard » du programme (la console).

Note: Ecrire dans la console

Pour écrire sur la console de sortie (pour écrire quelque chose dans une fenêtre spéciale de l'écran), on utilise cette syntaxe

```
1 System.out.println( <chaîne de caractères> ) ;
```

Pour l'instant cette expression semble très bizarre. On appelle la méthode `println` qui écrit dans la console le texte passé en paramètre. Cette méthode se trouve dans la classe `out` qui gère les sorties. Cette classe fait elle-même partie d'une classe nommée `System` qui gère l'entrée et la sortie standard, i.e. la saisie au clavier et l'affichage à l'écran. Nous reviendrons plus tard sur les entrées et sorties en JAVA (voir Section 7). Pour le moment, quand on utilisera ce code, cela affichera à l'écran le texte passé en paramètre.

L'exécution d'un programme va s'effectuer en deux étapes.

1. Tout d'abord, on va *compiler* le code source, c'est à dire on va traduire le code dans un langage intermédiare appelé le `byte code` et on va sauvegarder dans un fichier avec l'extension `.class`. Si la traduction se passe bien, on aura deux fichiers pour notre classe : notre fichier `HelloWorld.java` et la traduction en byte code `HelloWorld.class`.

2. Ensuite on va lancer la *machine virtuelle* qui va charger le fichier `.class` et exécuter le byte code.

Une fois compilé, le byte code peut tourner sur n'importe quelle machine virtuelle (sur un PC sous Windows, Os X, Linux, sur votre téléphone ou tablette, etc...). « write once, run anywhere ».

Si vous utilisez un environnement de programmation comme `Eclipse`, ces deux étapes sont transparentes : lorsque vous appuyez sur le bouton d'exécution, l'environnement lancera la compilation puis le lancement de la machine virtuelle. Si vous utilisez un éditeur de texte et un terminal, la compulation s'effectue avec le programme `javac` (java compilator). Le lancement de la machine virtuelle se fait à l'aide de la commande `java` (cd Figure 1.1)

2 Variables

2.1 Nom

Le nom d'une variable (il en est de même poru une méthode ou une classe), doit commencer par une lettre. Le nom peut contenir des lettres, des chiffres, ainsi que les symboles _ et $. A priori, π ou `déjàVue`) sont des noms valides (le compilateur saura les utiliser). Cela dit, pour échanger des codes sources (les fichiers `.java`), si des encodages différents sont utilisés, cela risque de poser problèmes, donc en pratique, utilisez des caractères non accentués.

2.2 Déclaration d'une variable

A chaque fois qu'on utilise une variable, par exemple `toto`, le compilateur JAVA doit savoir la nature de la variable, c'est ce qu'on appelle son *type*. Est-ce que `toto` est une valeur entière, ou bien une variable qui désigne un fichier ? Dans certains langage, les variables peuvent changer de type ou on n'a pas besoin d'explicitement déclarer le type de chaque variable. Il n'en est pas ainsi pour JAVA : en JAVA , toute variable doit être *déclarée* avant de pouvoir être utilisée, ce qui est bien normal, sinon le compilateur ne sait pas ce qu'il manipule ! Une variable ne peut pas changer de type. Pour ces raisons, on dit que le langage JAVA est *fortement typé*. Dans ce qui suit, on utilise la notation `<chose>` pour indiquer que dans le code il faudra un mot qui représente une `chose`. Par exemple `<type>` devra être remplacé par un type JAVA , `<nom>` sera remplacé par le nom d'une variable, etc. Le symbole `|` sera utilisé pour décrire un choix possible : par exemple
`<variable> | <expression>` veut dire qu'il faudra soit écrire le nom d'une variable, soit une expression JAVA complète.

Il y a plusieurs variantes pour déclarer des variables :

— *déclaration simple* : `<type> <nom>` ;
elle permet de déclarer une variable en indiquant tout d'abord son type puis son nom. Avec JAVA , il *n'y a pas* de valeur par défaut, toute variable doit donc être initialisée avant utilisation ! En utilisant cette variante de déclaration, il faudra dans le code initialisée la variable, ce que le compilateur de JAVA vérifiera.

— *déclaration avec affectation* :
`<type> <nom> = <valeur dans le type> | <variable> | <expression>` ;
On peut aussi déclarer une variable et l'initialiser en une seule instruction. Pour l'initialisation, on a le choix entre donner directement une valeur (qui bien sûr doit avoir le bon type), utiliser la valeur d'une autre variable, utiliser une expression dont l'évaluation aura le bon type.

— *déclaration multiple* : `<type>` $\texttt{<nom}_1\texttt{>, <nom}_2\texttt{>, ..., <nom}_k\texttt{>}$;
on peut déclarer plusieurs variables qui ont le même type dans la même instruction.

— *déclaration multiple avec affection partielle* : `<type>` $\texttt{<nom}_1\texttt{>,}$ $\texttt{<nom}_2\texttt{>= <valeur dans le type>,}$ $\texttt{..., <nom}_k\texttt{>}$;
lors d'une déclaration multiple, on peut initialiser certaines variables (pas forcément toutes).

Dans ce qui précède, le type peut être soit un type élémentaire présenté ci-dessous, soit de type complexe ou des objets. On parlera de ces objets plus tard.

2.3 Types élémentaires

JAVA fournit huit types élémentaires que peuvent prendre des variables. Lorsqu'une variable est déclarée à l'aide d'un de ces types, un espace (de taille correspondant au type) est automatiquement réservé en mémoire. Dans le tableau qui suit, on indique la taille mémoire en *bit*s [2] occupée par la variable.

type élémentaire	nombre de bits	interval de valeurs
`boolean`	1	deux valeurs `true` et `false`
`byte`	8	un entier compris entre -128 et 127
`short`	16	un entier compris entre -2^{15}=-32768 et $2^{15}-1$=32767
`int`	32	un entier compris entre -$2^{31} \approx -2.1 \cdot 10^9$ et $2^{31}-1 \approx 2.1 \cdot 10^9$
`long`	64	un entier compris entre -$2^{63} \approx -9.2 \cdot 10^{18}$ et $2^{63}-1 \approx 9.2 \cdot 10^{18}$
`char`	16	caractère unicode, il y a 65536 codes
`float`	32	nombre flottant norme IEEE
`double`	64	nombre flottant norme IEEE

2.4 Exemples d'affectation avec valeurs

```
1 short population ;
2 population = 30000 ;
```

Ici, on déclare un entier court (au plus 32,767) appelé `population` et on l'initialise avec 30,000. Que se passe-t-il si on utilise l'exemple suivant ?

```
1 short population = 1000000 ;
```

Ici, le compilateur vous indiquera que 1,000,000 est un entier, mais pas un short et vous obtiendrez une erreur. Maintenant, quel est le problème avec l'exemple suivant ?

```
1 long nbParticules = 10000000 ;
```

A priori 10,000,000,000 peut bien être encodé par un `long`, mais pas par un `int`. Cependant, cette ligne n'est pas correcte. Le problème qui se pose est comment le compilateur doit-il interpréter 10,000,000,000 ? En fait, la règle pour JAVA est qu'il interprète un nombre tapé comme un `int`. Le compilateur comprend ce nombre seulement comme un entier, mais il n'y parvient pas. Pour indiquer qu'il s'agit d'un long, il faut ajouter la lettre 'L' à la fin du nombre comme suit :

```
1 long nbParticules = 10000000L ;
```

de même pour les `float`s et les `double`s, on peut terminer un nombre avec `'f'` pour un `float` et avec `'d'` pour un `double`.

Pour les caractères, on doit utiliser les apostrophes comme suit (attention, un `char` correspond à un *seul* caractère, nous verrons plus tard comment faire une chaîne de caractères). JAVA utilise l'encodage `UTF-16`, un `char` est donc un code de cet encodage. Dans l'exemple suivant, on utilise la lettre 'c' qui a la valeur 99 en décimal (63 en hexadécimal), que l'on peut donc aussi écrie "\u0063". Vous pouvez donc utiliser des symboles plus exotiques (ex "\u23F0" est un réveil matin, `\uD83D \uDC19` une pieuvre).

Il y a quelques codes utiles : `''` est le retour à la ligne, `''` est une tabulation.

2. un bit est un espace mémoire qui peut prendre deux valeurs : 0 ou 1. Il faut 8 bits pour faire un *octet*

```
1 char lettre = 'c' ;
```

Finalement, le type `boolean` contient seulement deux valeurs `true` et `false`, ces deux mots sont des mots réservés du langage.

```
1 boolean test = true ;
2 test = false ;
```

2.5 Conversion

On a dit que chaque variable avait un type et qu'une variable ne pouvait pas changer de type. Cependant, on peut utiliser une variable d'un type pour initialiser une variable d'un autre type commensurable. On déclare avec affectation une première variable et on va l'utiliser pour initialiser une seconde variable. Si les variables ont le même type, il n'y a pas de problème, mais JAVA nous permet d'utiliser des types commensurables. Par exemple, on devrait pouvoir initialiser un `double` à l'aide d'un `int`. On pourrait aussi faire l'inverse, initialiser un `int` à l'aide un `float` en faisant une opération de conversion ou *cast*. On a donc la situation suivante ou l'on affecte une variable d'un certain type et où l'on utilise cette première variable pour réaliser l'affectation d'une seconde variable :

```
1 <type1> <nom1> = <valeur1> ;
2 <type2> <nom2> = <nom1> ;
```

La *conversion* ou *cast* peut rester *implicite* si le `<type1>` est « moins fort »que le `<type2>` : le `<type2>` va utiliser toute l'information du `<type1>` sans perte. Par exemple si le `<type1>` est un `int` et le `<type2>` est un `double`, le double utilisera la valeur de l'entier (et la partie après la virgule est initialisé à zéro). Dans ce cas, la ligne 2 du code ci-dessous n'entrainera aucune erreur de la part du compilateur.

```
1 int valeurEntiere = 17 ;
2 double valeurFlottante = valeurEntiere ;
```

La conversion doit devenir *explicite* si le `<type1>` est « strictement plus fort »que le `<type2>` : il faut indiquer au compilateur d'effectuer la conversion. Pour forcer la conversion, il faut indiquer le type cible de la conversion entre paranthèses comme suit :

```
2 <type2> <nom2> = (<type2>) <nom1> ;
```

La valeur du type « plus fort »est tronquée pour pouvoir prendre le type « plus faible ». Par exemple, pour convertir un `double` ou un `float` en un `int`, JAVA tronque la partie derrière la virgule. Dans l'exemple ci-dessous, la valeur de `valeurEntiere` sera de `3`.

```
1 double valeurFlottante = 3.141592 ;
2 int valeurEntiere = (int) valeurFlottante ;
```

3 Opérations arithmétiques

Les tableaux ci-dessous contiennent les différents opérateurs de JAVA . Les tableaux inclus la priorité de l'opérateur pour que les expressions ne soient pas ambigues. On peut bien sûr ajouter des parenthèses pour qu'une expression complexe soit plus lisible.

Opérateur	priorité	action	exemples
+	1	signe positif	`+a ; +7`
-	1	signe négatif	`-a ; -(a-b) ; -7`
!	1	négation logique	`!(a<b) ;`
++		assignement et incrément de 1	`n++ ;++n ;`
--		assignement et incrément de 1	`n++ ; --i ;`

Opérateurs unaires

Opérateur	priorité	action	exemples
*	2	multiplication	`a * i`
/	2	division	`n/10`
%	3	reste de la division entière	`k%n`
+	3	addition	`1+2`
-	3	soustraction	`x-5`
<	5	strictement inférieur	`i<n`
<=	5	inférieur ou égal	`i <= n`
>	5	strictement supérieur	`i < n`
>=	5	supérieur ou égal	`i >= n`
==	6	égalité	`i==j`
!=	6	différent	`i !=j`
&	7	conjonction (et logique)	`(i<j) & (i<n)`
|	9	disjonction (ou logique)	`(i<j) | (i<n)`
&&	10	conjonction optimisée	`(i<j) && (i<n)`
||	11	disjonction optimisée	`(i<j) || (i<n)`

Opérateurs binaires

L'affectation d'une variable peut être fait à l'aide d'une expression. Pour certaines expressions très utilisées (incrémente un entier d'une unité par exemple), JAVA propose des raccourcis :

— `++x` incrémente `x` de 1 puis utilise sa valeur pour l'expression
— `x++` utilise la valeur de `x` pour l'expression puis incrémente `x`
— `i += 5` est le raccourci de `i = i+5`

```
1 int i=2, j = i++;
2 i=2;
3 j= ++i;
```

Après l'exécution de la ligne 1, `i` vaut 3 et `j` vaut 2. Après l'exécution de la ligne 3, `i` et `j` valent 3.

Une erreur très classique est l'utilisation du symbole = à la place de == pour l'égalité. Pour JAVA , l'expression `i=k` a bien un sens car c'est l'affectation de la variable `i` avec le contenu de la variable `j`. Etonnement, cette expression a aussi une valeur : c'est un `int` qui est positif si l'affectation se déroule bien, et négatif sinon. Un `int` pouvant être interprété comme un boolean, JAVA fait une conversion implicite. Ainsi, une ligne de code `(i=j)` peut être interprétée comme un booléen.

Il existe un opérateur conditionnel ternaire qui affecte la valeur d'une variable en fonction d'un test :

```
1 result = someCondition ? value1 : value2;
```

Si le test (une expression booléenne) `someCondition` est vérifié, alors la variable `result` prend la valeur `value1`, sinon elle prend la valeur `value2`. Dans l'exemple ci-dessous, `absX` prend pour valeur celle de la valeur absolue de `x`.

```
1 absX = (x > 0) ? x : -x ;
```

3.1 Type d'une expression

Lorsque les opérandes d'une expression ne sont pas du même type, il faut savoir quel est le type de la valeur de l'expression. La méconnaissance de ces règles entraine de nombreux bugs.

```
1 int i = 5,j ;
2 double x = 5.0 ;
3 j=i/2 ;
4 j=x/2 ;
5 double z = i/2 ;
```

L'affectation de la ligne 3 est bien correcte : `i/2` est une division entre deux `int`, c'est donc un `int`, et donc, on peut se servir de la valeur pour l'affectation de l'`int j`. L'affectation de la ligne 4 est quant à elle incorrecte : `x/2` est une division entre un `double` et un `int`, c'est donc un `double`. Comme on essaye d'affecter la valeur d'un `int`, il faut utiliser le transtypage explicite. La ligne 5 est correcte, mais quelle est la valeur de `z` ? `i` est un `int`, `2` aussi, donc on a une division entre deux `int` dont le résultat est un `int` ! On a donc la division entière entre 5 et 2, dont le résultat est 2. Cette valeur est donc stockée dans le `double z`, donc `z` a pour valeur `2.0`. Si on voulait obtenir 2.5, une astuce est de forcer une division entre un entier et un double par exemple en utilisant l'expression :

```
5 double z = i/2.0 ;
```

Le transtypage explicite peut être vu comme un opérateur unaire qui a une priorité haute. Lors des opérations de transtypage explicite, il faut bien faire attention à la portée du transtypage, i.e. sur quelle expression porte le transtypage.

```
1 double x = 2.75 ;
2 int y = (int) x * 2 ;
3 int z = (int) (x * 2) ;
```

La valeur de `y` après l'exécution de la ligne 2 sera 4 car la conversion porte simplement sur `x`, et donc la multiplication s'opère entre `int`. La valeur de `z` après exécution sera 5 car la conversion porte sur le résultat de la multiplication entre un `double` et un `int`, qui est un `double`.

4 Décision

JAVA propose des structures très classiques pour effectuer certaines instructions en fonction de la valeur de certaines variables.

4.1 Branchements

Condition

La structure `if ... then ... else`

Cette structure permet de choisir d'exécuter un code différent selon une condition : si une condition est remplie, on exécute un bloc d'instructions, sinon on exécute un autre bloc d'instructions. La structure est la suivante :

```
if ( <expression booléenne> )
   <bloc d'instructions à exécuter si la condition est satisfaite>
else
   <bloc d'instruction à exécuter si la condition n'est pas satisfaite>
```

La partie « sinon ». i.e. le `else` et le bloc d'instruction qui suit est optionnel, on peut juste donc avoir une structure si une condition est remplie, alors on exécute un bloc d'instructions.

Dans la structure présentée ci-dessus, on utilise des blocs d'instructions, donc a priori après la condition, on ouvre une accolade pour le bloc d'instruction et on le ferme juste avant le `else`. Si un bloc est composé d'une seule instruction, on n'a pas besoin d'utiliser les accolades.

```
int gains,payment,encaissement ;
...
// opérations qui modifient la variable gains
...
if (gains>0)
  encaissement = gains ;
else
  payment = gains ;
```

On peut imbriquer une autre structure de condition dans une structure existente, de cette façon, on peut utiliser plusieurs tests : si une condition est satisfaite, on fait cela, *sinon si* une autre condition est satisfaire, on fait ceci, etc.

```
int gains,payment,encaissement,investissement ;
...
// opérations qui modifient la variable gains
...
if (gains<0)
  payment = gains ;
else if (gains > 10) {
  encaissement = 10 ;
  investissement = gains-10 ;
}
else
  encaissement = gains ;
```

On a simplement indiqué que la condition devait être une expression bouléenne. Dans l'exemple ci-dessus, elle est simple et formée d'une seule condition. Bien sûr, on peut écrire des conditions plus complexes à l'aide de conjonctions (« et » logique représenté par `&&`) et de disjonctions (« ou » logique représenté par `||`). Attention, le compilateur vérifiera seulement si la condition est une expression booléenne, pas si votre test à un sens (par exemple vous écrivez une tautologie ou un test qui sera toujours faux !).

choix multiples

Si vous voulez exécuter un bloc de code différent selon les valeurs d'une variable, vous pouvez utiliser une structure appelée `switch`. Par exemple, pensez au menu quand vous appelez une société : tapez 1 si vous voulez ceci, tapez 2 si vous voulez cela, etc. La structure se présente comme dans l'exemple

ci-dessous :

```
int choix ;
...
// l'utilisateur modifie la valeur de choix
...
switch(choix) {
  case 1 :
    //instructions pour le choix 1
    ...
    break ;
  case 2 :
    //instructions pour le choix 2
    ...
    break ;
  default
    // instruction dans le reste des cas
    ...
}
```

On indique après le mot clé `switch` la variable sur laquelle on va faire le choix. Dans l'exemple, c'est la variable `choix`. Ensuite, chaque cas commence par le mot clé `case` suivi d'une valeur. Ici, on a deux cas selon si la variable `choix` a pour valeur 1, 2, et le cas `defaut` indique ce que l'on doit faire dans tous les autres cas. Chaque cas se termine par le mot clé `break`. L'instruction `break` permet de sortir de l'expression en cours. Ici, utilisée à l'intérieur du `switch`, elle permet de sortir du switch lorsqu'un des choix correspond au cas en cours. Si vous oubliez le `break`, plusieurs cas vont être exécutés au lieu du seul cas qui convient ! !

On peut faire un `switch` sur deux types de variables : soit sur un `int` comme dans l'exemple, soit sur un `char`. Depuis la version 7 de JAVA , on peut aussi utiliser une chaîne de caractères, que nous verrons plus tard.

4.2 Itérations

On présente maintenant trois moyens de faire des boucles.

boucle for

Une boucle « `for` »est généralement utilisée lorsque l'on sait combien de fois on va réaliser l'itération. La structure de la boucle est comme suit :

```
for (<initialisation> ; <condition de fin> ; <mise à jour des valeurs>)
  <bloc d'instructions>
```

S'il y a une seule instruction dans la boucle, on peut omettre les accolades. Il est bon d'indenter son code pour voir clairement le code qui sera itéré. Attention, il n'y a pas de point-virgule juste à la fin du `(for ...)` ! Dans la partie d'initialisation, on peut avoir plusieurs déclarations et initialisations de variables. La condition de fin est une expression booléenne (par exemple, on peut avoir des conjonctions ou des disjonctions d'expressions de tests). On peut mettre à jour plusieurs valeurs à la fois. Ci-dessous voici quelques exemples :

```
1 for ( ; ; ){
2     // some instructions
3 }
```

Q : Que se passera-t-il ?

L'exemple le plus classique est le suivant :

```
0 int n=10 ;
1 for (int i=0 ; i< n ; i++ ){
2     // some instructions
3 }
```

On rappelle que `i++` est équivalent à `i=i+1`. Si par exemple on a un tableau de taille de dimension 1 et de taille `n`, on peut utiliser une boucle comme celle-ci pour faire une opération sur chaque élément du tableau.

```
0 int n=10 ;
1 for (int i=0, j=n ; j< i ; i++ ; i-- ){
2    // some instructions
3 }
```

Dans cet exemple, on utilise deux variables pour la boucle.

On verra un peu plus tard une autre syntaxe pour les boucles `for` (Section 2.2)

boucle while

Cette structure est généralement utilisée quand on ne sait pas combien de fois on doit itérer, le coeur de la boucle est donc le test d'arrêt de la boucle. La structure se comprends comme suit : tant que la condition d'arrêt est satisfaite, on continue l'itération. La boucle s'écrit comme suit :

```
1 while(<condition de fin>)
2    <bloc d'instructions>
```

Voici un exemple qui va essayer de déterminer une approximation de la valeur de la limite de la suite convergente $u : n \to 0.75^n$:

```
1 double epsilon = 0.0000001 ;
2 double r = 0.75, u=1 ;
3 while( u - u* r > epsilon)
4     u = u * r ;
```

boucle do while

La structure de cette boucle est similaire à une boucle `while`. Au lieu de débuter par tester la condition d'arrêt avant d'effectuer une itération, on commence ici par faire une itération avant de tester la condition d'arrêt. La boucle est naturelle lorsqu'on peut exprimer le code par une phrase comme « *on fait ce bloc d'instructions tant que la condition est vérifiée* ». La structure de la boucle est comme suit :

```
1 do
2     <bloc d'instructions>
3 while(<condition de fin>) ;
```

Attention de ne pas oublier le point-virgule à la fin du `while`. On peut donc écrire l'exemple de la boucle `while` à l'aide de la boucle `do ... while ()`.

```
1 double epsilon = 0.0000001 ;
2 double r = 0.75, u=1 ;
3 do
4     u = u * r ;
5 while ( u - u* r > epsilon) ;
```

Quelle boucle choisir ?

Selon les cas une des structure est plus élégante que les autres. Si vous connaissez le nombre d'itérations nécessaires, généralement une boucle `for` est préférable. Sinon utilisez une boucle `while` ou `do ... while`.

On peut utiliser l'instruction `break` pour sortir de l'instruction courante, donc pour sortir d'une boucle `for`. Certains utilisateurs utilisent donc cette fonctionalité pour écrire une boucle `for` là où une boucle `while` serait plus adéquate. Je recommande d'utiliser une boucle `while` car la condition de fin est mise en valeur. Comme cette condition est une source d'erreurs, elle est bien plus facile à identifier. Au contraire, si quelqu'un lit un code et voit une boucle `for` sans chercher à lire exactement le bloc d'instructions, il peut penser que le code sera itéré un nombre de fois précis, alors que ce n'est pas le cas à cause du break.

Dans l'exemple suivant, on veut chercher si un élément `k` se trouve dans un tableau d'entier, et si oui, de trouver son index dans le tableau. Le code suivant est correct.

```
1 int[] tableau = new int[10] ;
2 // remplissage du tableau
3 for ( int index = 0 ; i< tableau.length ;index++) {
4   if(tableau[index] == k)
5     break ;
6 }
```

Cependant, on ne sait pas a priori combien de fois on va itérer le test « est-ce que le i^{ime} élément est égal à `k` », on peut donc utiliser une boucle `while` comme suit :

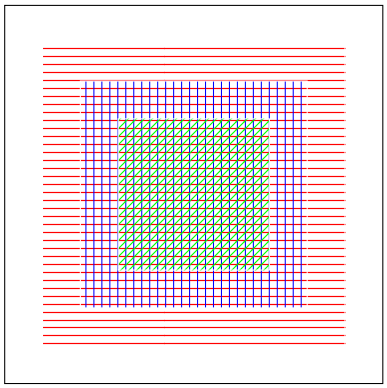
```
1 int[] tableau = new int[10] ;
2 // remplissage du tableau
3 int index = 0 ;
4 while(index < 10 && tableau[index] != k)
5     index++ ;
```

Ces deux codes sont équivalents au niveau de la performance. Peut-être que le second est plus facile à relire.

4.3 Visibilité d'une variable dans un bloc

On peut délimiter des blocs d'instructions en les délimitant par `{` et `}`. Ainsi, on peut avoir des blocs imbriqués les uns dans les autres. On verra des utilisations particulières de blocs dans la section suivante. Mais ce qu'il est important de savoir est que les variables déclarées dans un bloc interne ne sont pas connues dans un bloc plus externe ! C'est parfois pratique lorsqu'on utilise des variables auxiliaires pour faire un calcul.

```
1  int a,b=10 ;
2  {
3    int d=2*b ;
4    {
5      int e=b+d ;
5      a=e*d ;
6      {
5        int g= b+ d*e ;
6      }
6    }
7  }
```



`a` et `b` sont connus partout
`d` est connu seulement dans la partie rouge
`e` est connu seulement dans la partir blue
`g` est connu seulement dans la partie verte

5 Méthodes

Dans un projet, on a souvent de faire les mêmes opérations. On peut donc choisir de mettre le code correspondant dans une fonction. En JAVA , on ne parle pas de fonctions, mais de *méthodes*. L'intérêt d'utiliser des fonctions est double :

— en appelant la fonction, on diminue la taille du code et le code devient plus lisible.
— si on veut modifier le code (par exemple changer l'implémentation pour améliorer les performances), il n'y a plus qu'un seul endroit à changer.

5.1 Déclaration

On parlera beaucoup plus de méthodes dans la partie de programmation orientée objet. Dans cette section, on va présenter un type de méthodes qui suit la structure suivante :

```
1  public static <type de retour> <nom>( <liste de paramètres>) {
2     corps de la méthode : suite d'instructions
3  }
```

On expliquera `public` et `static` un peu plus tard.

Une méthode peut retourner une valeur, comme une fonction en mathématique. Il faut donc indiquer le type de valeurs qui est retournée par la méthode. Cela peut être un type simple ou un type complexe comme un objet. Si la méthode ne retourne rien, on indique alors que la méthode retourne `void`.

Une méthode a bien sûr un nom et c'est ce nom qui sera utilisé lors de l'appel de la méthode. Lorsque vous le choisissez, soyez créatif pour donne un nom adéquat qui donnera du sens quand vous appelerez la méthode.

Enfin, on place entre parenthèses la listes des arguments de la méthode. Ce sont les paramètres dont la méthode a besoin pour remplir sa tâche. Il n'y a pas de limite du nombre de paramètres, et on peut aussi avoir des méthodes sans paramètres (dans ce cas, il faut quand même écrire les parenthèses ouvrante et fermante). L'ordre des arguments joue un rôle important et doit être respecté lors de l'appel de la méthode.

Le nom de la méthode et la liste ordonnée des arguments constituent la *signature* de la méthode. Cette signature doit être unique, i.e., on ne peut pas avoir une autre fonction avec le même nom et la même liste d'arguments [3].

3. Ceci n'est pas complètement exact. On verra qu'avec les espaces de noms (Section 3) on peut avoir deux méthodes avec

Finalement, après cette déclaration de la méthode suit le bloc qui contient le corps de la méthode, i.e. les instructions qui doivent être exécutées. Si la méthode retourne une valeur, la méthode doit se terminer par une instruction `return` qui va renvoyer la valeur qui suit.

```
k | return <val> ;
```

Attention, une fois que cette instruction est exécutée, on va quitter la méthode et les instructions qui pourraient se trouver en dessous de l'expression `return` ne seront pas exécutées.

Dans l'exemple ci dessous, on va écrire une fonction qui va chercher l'élément maximum d'un tableau de `int`, elle va donc retourner un `int` et va prendre comme paramètres un tableau de `int`.

```
1 | public static int max( int[] tableau) {
2 |   int m= tableau[0] ;
3 |   for (int i=1 ;i<tableau.length ; i++){
4 |     if (tableau[i] > m)
5 |       m = tableau[i] ;
6 |   }
7 |   return m ;
8 | }
```

On peut avoir une fonction avec le même nom mais une liste d'arguments différents. Par exemple, on peut facilement faire une autre fonction `max` qui retourne le maximum d'un tableau de `double`. Cela s'appelle *surcharger une méthode*. Lors de l'exécution, JAVA choisira la méthode appropriée selon le type des arguments.

Pour appeler la fonction, il suffit simplement d'utiliser son nom et de remplir la liste des arguments en respectant bien l'ordre des arguments et leurs types. Par exemple dans l'exemple suivant, on appelle la méthode `max` que l'on vient de créer.

```
1 | int tab = {7, 12, 15, 9, 11, 10, 17, 9, 13} ;
2 | int m = max(tab) ;
```

En JAVA , les arguments sont toujours passés *par valeurs*. Pour les types primitifs, ce passage se fait par *copie*, c'est à dire que si on utilise une variable comme argument dans une méthode, la valeur de la variable est copiée en paramètre. La variable elle même ne sera pas affectée par l'appel de la méthode. Pour un type complexe, i.e., pour un objet ou un tableau, le passage se fait par référence, on pourra donc modifier un objet en utilisant une méthode. Dans l'exemple suivant, après l'exécution de la ligne 10, chaque entrée du tableau `tab` contient `0`, par contre la valeur de `a` sera toujours de `100`.

des signatures identiques, mais sans que cela pose des problèmes d'ambiguïté.

```java
public static void zeros( int[] tableau, int a) {
   for (int i=0 ;i<tableau.length ; i++){
     tableau[i] =0 ;
     a = 0 ;
}

public static void main(String[] arg){
   int[] tab = {10, 20, 30, 40, 50} ;
   int a = 100 ;
   zeros(tab,a) ;
}
```

Si vous avez déjà codé en C, vous pouvez déjà coder beaucoup de choses en utilisant la syntaxe JAVA présentée jusqu'ici. Dans le chapître suivant, on va commencer à présenter la particularité du langage JAVA : l'orientation objet.

5.2 une méthode particulière : la méthode `main`

Pour exécuter une application ou un projet, il faut appeler une méthode spéciale : la méthode `main`. Ainsi, dans l'ensemble des classes d'un projet, il faut au moins avoir défini une méthode `main` qui lancera le projet. Sa signature est fixée comme suit (si vous ne respectez pas parfaitement cette signature, JAVA ne pourra pas comprendre que cette méthode est la « véritable « méthode `main`.) :

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

- la méthode est publique car c'est elle que l'on va appeler pour lancer l'application
- la méthode est statique. Heureusement, car sinon il faudrait déjà créer une instance, et comme `main` lance l'application, il serait difficile de créer un objet avant cela !
- la méthode ne renvoie rien, ce qui n'est pas surprenant car on se demande à qui elle renverrait une information.
- `main` est le nom de la méthode
- elle a pour seul argument un tableau de `String`. En effet, lorsqu'on lance un programme en JAVA , on tape une commande qui lance l'exécution. On peut ajouter du texte à la fin de la commande et chaque mot ajouté est inséré dans le tableau `args`. Ces mots pourront être utilisés comme option pour l'application.

Dans le code ci-dessous, l'exécution de la méthode `main` affichera Bonjour, Hello ou Hola selon le premier mot passé en option.

```java
public static void main(String[] args){
   switch(args[0]){
     case "francais" :
       System.out.println("Bonjour") ;
       break ;
     case "english" :
       System.out.println("Hello") ;
       break ;
   }
}
```

L'exemple ci-dessous montre deux exécutions :

```
~$ java introjava/HelloWorld francais
Bonjour
~$ java introjava/HelloWorld english
Hello
```

6 Une structure de données : déclarer et utiliser des tableaux

JAVA propose une stucture de données particulière pour faire des tableaux à n dimensions. Pour déclarer un tableau, on indique le type du tableau suivi d'autant de `[]` que le tableau a de dimension. Dans l'exemple suivant, `ligne` est un tableau à une dimension, `rectangle` un tableau a deux, et `cube` à trois.

```
<type> [] ligne ;
<type> [][] rectangle ;
<type> [][][] cube ;
```

Pour initialiser un tableau, il faut créé cette structure en mémoire (i.e., il faut réserver de l'espace mémoire qui contiendra le tableau). On utilise le mot clé `new` qui va effectuer cette création (on verra une utilisation similaire pour créer un nouvel objet), puis on spécifie de nouveau le type et le nombre d'éléments dans chaque dimension. Donc à la création, il faut connaître la taille maximale du tableau !

```
<type> [] ligne = new <type>[<taille1>] ;
<type> [][] rectangle = new <type>[<taille2>][<taille3>] ;
<type> [][][] cube = new <type>[<taille4>][<taille5>][<taille6>] ;
```

Lors de la création d'un tableau, celui-ci est automatiquement initialisé avec une valeur par défaut. La valeur par défaut est

- 0 pour des tableaux contenant des types numériques (`char` inclus)
- `false` pour un tableau de `boolean`
- `null` pour un tableau contenant des objets

Pour accéder aux éléments du tableau, on utilise le nom du tableau, et les indexes dans chaque dimension. Par exemple `rectangle[3][4] + cube[1][2][5] ;`. On peut obtenir la taille du tableau en utilisant `.length` comme dans l'exemple suivant :

```
int n = ligne.length ;
```

On peut avoir en tête que `[]` s'applique à un type et que `<type>`, `<type>[]`, et `<type>[][]` sont des types. Donc on peut écrire le code suivant :

```
1 int[][][] cube = new int[3][4][5] ;
2 int[][] rectangle = cube[2] ;
3 int n1 = cube.length ;
4 int n2 = cube[0].length ;
5 int n3 = cube[0][0].length ;
```

Ainsi, rectangle sera un moyen pour accéder aux éléments `cube[2][i][j]` ; `n1` contiendra 3, `n2` contiendra 4 et `n3` contiendra 5.

Attention le premier élément d'un tableau a pour index `0`, et donc le dernier élément a pour index `length-1`.

Pour initialiser un tableau, on peut aussi écrire les éléments dans l'ordre entre accolades. Dans ce cas là, JAVA réserve l'espace mémoire approprié et effectue l'initialisation (on n'a pas besoin d'utiliser **new** dans ce cas là). Dans l'exemple qui suit, on initialise un tableau à une dimension et un tableau à deux dimensions.

premiers :

2	3	5	7	11	13	17	19	23

triangle :

1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1

```
1 int[] premiers = {2, 3, 5, 7, 11, 13, 17, 19, 23 } ;
2 int[][] triangle = {{1,1,1,1}, { 0,1,1,1}, { 0, 0, 1, 1}, {0, 0, 0, 1} } ;
```

Lorsqu'un nouveau tableau est créé, il occupe un espace mémoire. Dans l'exemple ci-dessous, on définit un tableau `premier` et on déclare une autre variable de type tableau d'entiers. En utilisant l'opérateur =, on donne un nouveau nom au même espace mémoire, mais on ne fait pas une copie du tableau. Dans l'exemple ci-dessous, il n'y a qu'un seul tableau en mémoire, donc à la fin de l'exécution, on aura `premiers[2]=4`.

```
1 int[] premiers = 2,3,5,7,11 ;
2 int[] entiers ;
3 entiers= premiers ;
4 entiers[2]=4 ;
5 System.out.println(premiers[2]) ;
```

index	valeur
0	2
1	3
2	~~5~~ 4
3	7
4	11

On verra plus tard que JAVA offre des moyens simples pour faire des opérations sur les tableaux. Nous présentons des examples maintenant, sans expliquer ce qui se passe, cela sera fait Section **??**

— `Arrays.toString` permet d'obtenir la représentation d'un tableau en chaîne de caractères
— `Arrays.sort` trie un tableau
— `Arrays.BinarySearch` permet de chercher un élément dans un tableau

```
1 int[] tab = 4, 7, 3, 1, 2 ;
2 System.out.println("tab : " + tab) ;
3 System.out.println("tab : " + Arrays.toString(tab) + "(toString)") ;
4 Arrays.sort(tab) ;
5 System.out.println("trie : " + Arrays.toString(tab)) ;
6 System.out.println("index de 4 : " + Arrays.binarySearch(tab,4)) ;
```

L'exécution du code ci-dessus produira le résultat suivant :

```
tab : [I@7852e922
tab : [4, 7, 3, 1, 2](toString)
trie : [1, 2, 3, 4, 7]
index de 4 : 3
```

Ce qui est affiché lors du premier affichage est en fait l'adresse mémoire du tableau (on peut comprendre l'ambiguité : `tab` désigne-t-il la première case mémoire du tableau ou le tableau en entier). On peut donc appeler la méthode `Arrays.toString` pour afficher les valeurs contenues dans le tableau.

7 Enumération

Parfois, on a besoin d'une liste de valeurs possibles, par exemple, les tailles des vêtements sont souvent le valeurs suivantes : `XS`, `S`, `M`, `L`, et codeXL. On pourrait évidemment utiliser ces symboles et leur associer une valeur (par exemple un `int` (on verra même plus tard comment on pourrait s'assurer que la valeur associée ne puisse être changée). Mais JAVA offre un mécanisme pour directement utiliser un type énuméré.

Pour créer un type énuméré, on va déclarer dans un fichier source JAVA le nom du type énuméré puis les valeurs possibles. Pour créer le type énuméré `Size`, on va donc écrire dans le fichier `Size.java` le code ci-dessous :

```
1 public enum Size {
2    XS,
3    S,
4    M,
5    L,
6    XL
7 }
```

Pour l'utilisation, on pourra maintenant utiliser `Size` comme un type dont les valeurs sont `Size.XS`, `Size.S`, `Size.M`, `Size.L`, et `Size.XL`. On pourra aussi utiliser la méthode `values()` qui retourne la liste de valeurs possibles. Ceci permet d'utiliser une boucle for comme suit :

```
1  public class Exemple{
2     public static void main(String[] args){
3       Size mySize = Size.M ;
4       for (Size s : Size.values()){
5         if (s==mySize)
6           System.out.println("It is my size : "+s) ;
7         else
8           System.out.println(s + " is not my size") ;
9       }
10    }
11 }
```

L'exécution du code précédent donnera :

```
XS is not my size
S is not my size
It is my size : M
L is not my size
XL is not my size
```

Notez que le type énuméré est bien adapté pour faire un `switch`

```
1  public class Exemple{
2     public static void main(String[] args){
3       Size mySize = Size.M ;
4       double price =0 ;
5       switch(mySize){
6         case S : price = 5 ; break ;
7         case M : price = 7 ; break ;
8         case L : price = 9 ; break ;
9         case XL : price = 10 ; break ;
10        }
11      System.out.println("the price is : " + price) ;
12    }
13 }
```

La méthode `ordinal` permet de savoir la position d'une valeur dans la liste de valeurs (en partant de 0). Ceci permet d'avoir une comparaison entre les valeurs. On verra dans le chapitre 8 quelques options plus avancée.