



Programmation Java

Partie 1 – Algorithmique basique

Plan du chapitre

1. Installation
2. Premier programme
3. Les variables
4. Programmes dynamiques
5. Structures conditionnelles
6. Les Arrays
7. Structures itératives



Chapitre 1



Installation

Téléchargement

Installation

Variable d'environnement

Premier script

Compilation et exécution

Téléchargement

Téléchargez la dernière version de java sur son site officiel :
<https://www.oracle.com/java/technologies/downloads/#java19>

Java SE Development Kit 19.0.1 downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications and components using the Java programming language.

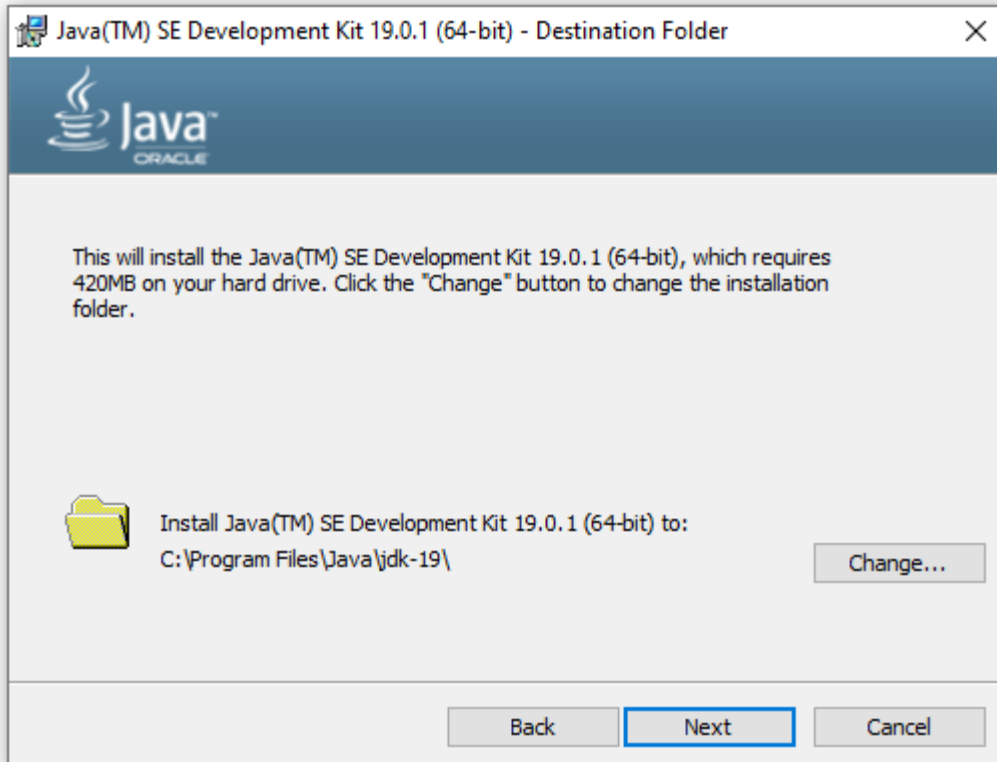
The JDK includes tools for developing and testing programs written in the Java programming language and running on the Java platform.

Linux	macOS	Windows	
Product/file description		File size	Download
x64 Compressed Archive		179.13 MB	https://download.oracle.com/java/19/latest/jdk-19_windows-x64_bin.zip (sha256)
x64 Installer		158.91 MB	https://download.oracle.com/java/19/latest/jdk-19_windows-x64_bin.exe (sha256)
x64 MSI Installer		157.76 MB	https://download.oracle.com/java/19/latest/jdk-19_windows-x64_bin.msi (sha256)

*Programme d'installation
pour Windows*

La version **19.0.1** sera utilisée lors de ce cours. Il s'agit d'une version encore en cours de développement, mais d'autres versions peuvent également être utilisées sans soucis comme la 17 par exemple.

Installation



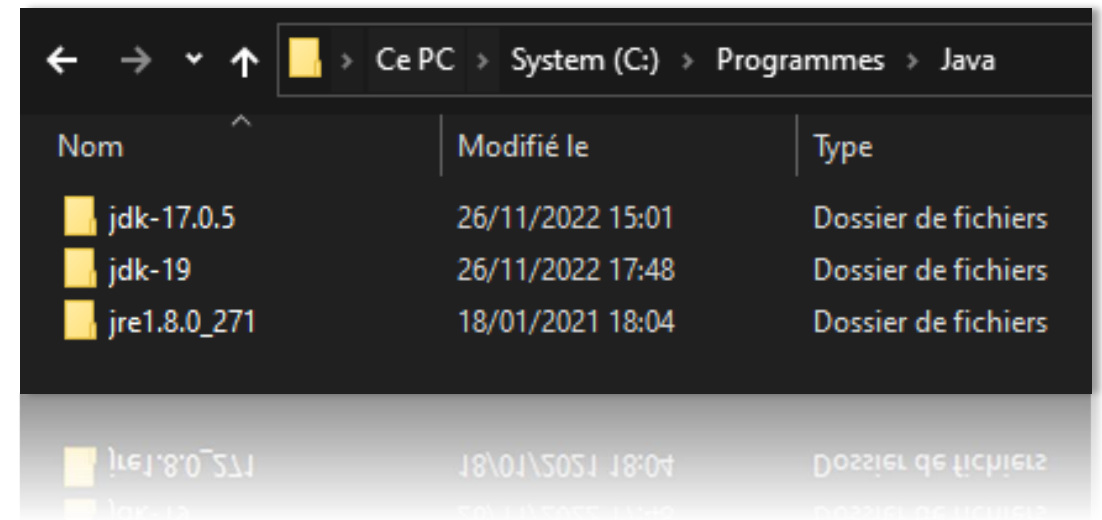
Exécutez ensuite l'installateur pour procéder à l'installation au travers de quelques étapes simple.

Il est conseillé de ne rien changer au réglages par défaut, une installation la plus basique est suffisante.

Installation

Vous devriez alors retrouver vos différentes versions dans <C:/Programmes/Java> pour une installation classique sur Windows.

Notez que pour avoir accès aux fonctionnalités de notre version de Java, il nous sera nécessaire de renseigner le dossier **bin** de l'installation dans nos [variables d'environnement](#).



Environnement de développement

Nous allons à présent mettre en place notre environnement de développement, c'est-à-dire les outils nécessaires au développement d'une application Java.

Les deux outils essentiels en Java sont **un éditeur de texte** qui nous permettra d'écrire notre code et **un terminal** dans lequel nous exécuterons notre programme.

Environnement de développement - *Editeur*

Dans ce cours, nous utiliserons **VSCode** pour éditer nos programmes. Il s'agit d'un éditeur très polyvalent et gratuit téléchargeable ici : <https://code.visualstudio.com/download>

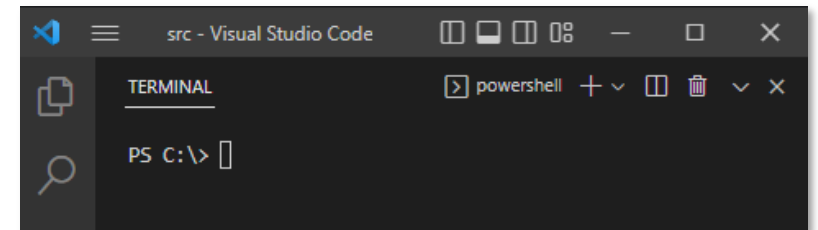
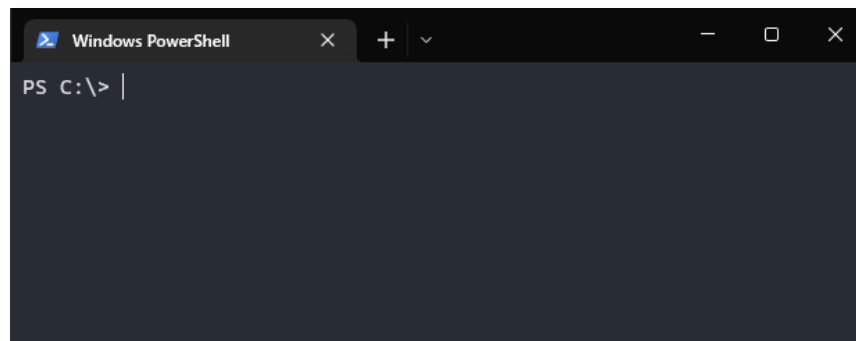
Cependant, il existe des outils plus simples comme **notepad++** ou **SublimeText**, ou des outils plus spécialisés Java comme **Eclipse** ou **NetBeans**.

Libre à vous d'essayer et de choisir votre logiciel.



Environnement de développement - *Terminal*

En ce qui concerne le terminal, le choix est encore plus simple. Dans ce cours, nous utiliserons le terminal intégré à **VSCode** mais il est tout à fait possible d'utiliser le terminal natif de votre système.



Environnement de développement - *Vérification*

Une fois votre choix de terminal effectué, nous allons vérifier que deux programmes y soient bien accessible pour commencer notre développement.

Effectuez les commandes suivantes pour obtenir la version des programmes **java** et **javac** :

- **java -version**
- **javac -version**

```
TERMINAL
PS C:\> java -version
java version "19.0.1" 2022-10-18
Java(TM) SE Runtime Environment (build 19.0.1+10-21)
Java HotSpot(TM) 64-Bit Server VM (build 19.0.1+10-21, mixed mode, sharing)
PS C:\> javac -version
javac 19.0.1
PS C:\> 
```

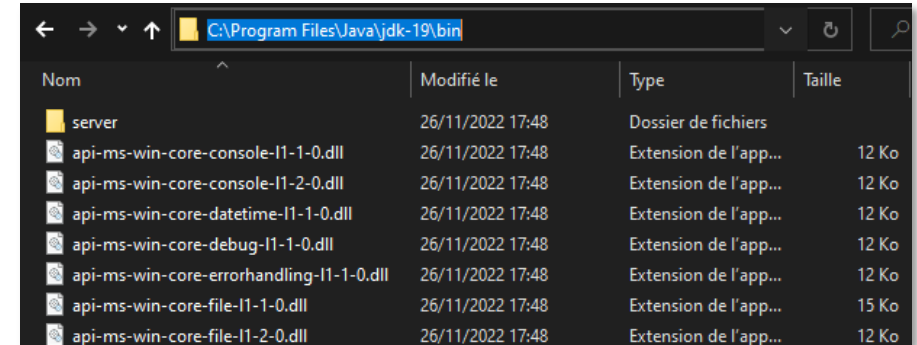
Variable d'environnement

Selon l'installation réalisée, ces commandes peuvent ne pas être disponible depuis le terminal. Nous allons devoir indiquer à Windows le ou les dossiers dans lesquels elles se trouvent.

```
PS C:\Users\Admin> java
java : Le terme «java» n'est pas reconnu comme nom d'applet de commande, fonction, fichier de script ou programme exécutable. Vérifiez l'orthographe du nom, ou si un chemin d'accès existe, vérifiez que le chemin d'accès est correct et réessayez.
Au caractère Ligne:1 : 1
+ java
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: ( java:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

Variable d'environnement

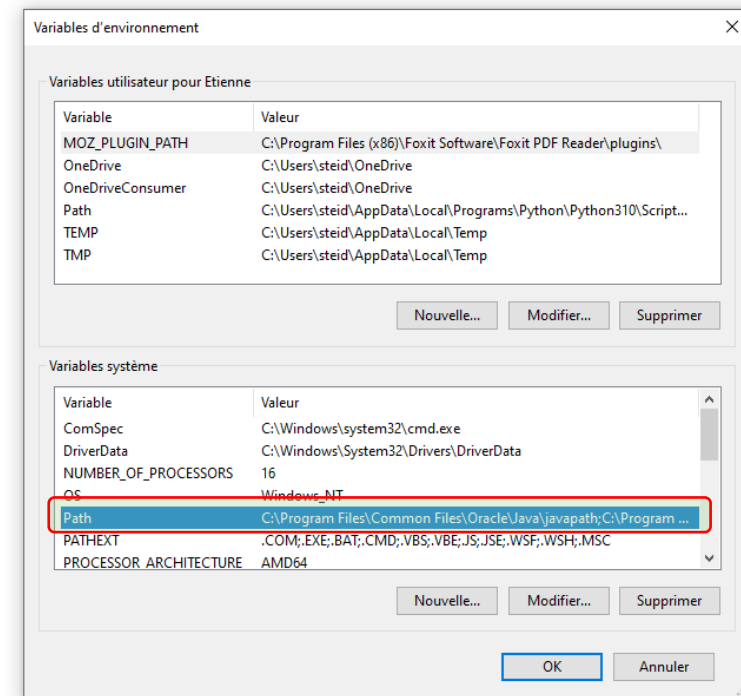
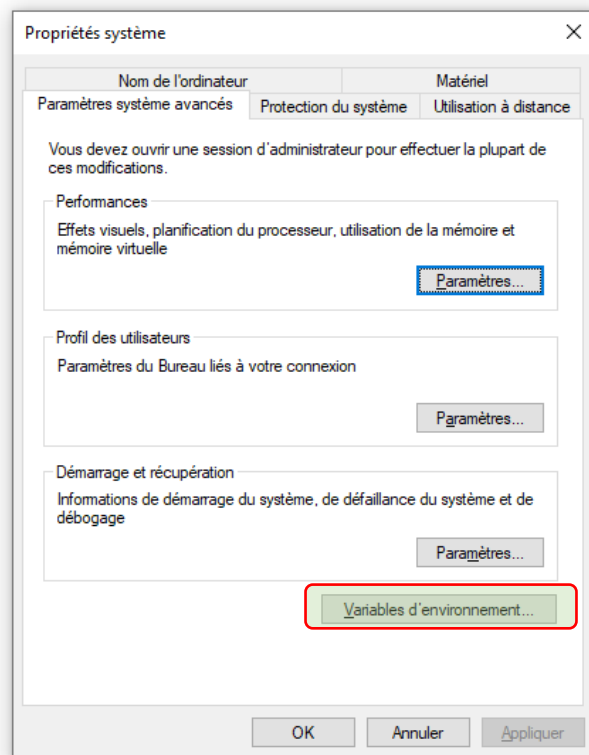
Pour ce faire, rendez-vous dans le dossier bin de votre installation de Java et copiez y l'adresse du dossier.



Dans la barre de recherche Windows, recherchez [Modifier les variables d'environnement système](#).

Variable d'environnement

Cliquer sur le bouton [Variables d'environnement](#) en bas à droite.



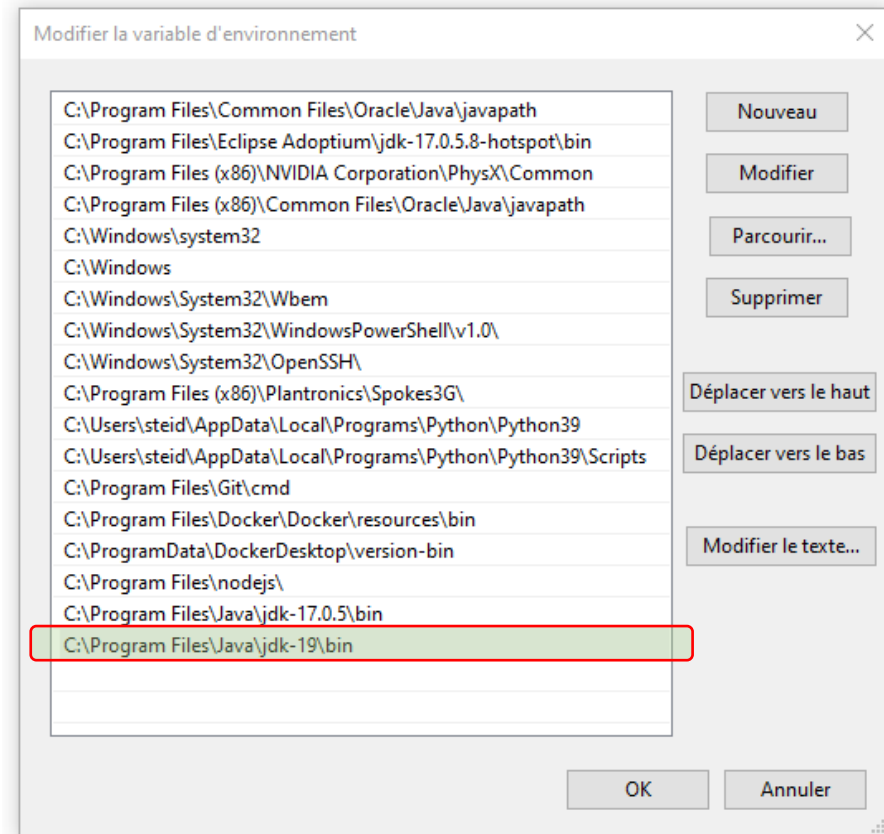
Puis double-cliquer sur la ligne [Path](#) dans la section [Variables système](#).

Variable d'environnement

Enfin, dans la nouvelle fenêtre [Modifier la variable d'environnement](#), ajouter une nouvelle ligne dans laquelle copier l'adresse du dossier bin de votre installation Java.

Pour ce faire, vous pouvez soit double-cliquer sur une nouvelle ligne, soit cliquer sur le bouton [Nouveau](#) en haut à droite.

Attention à ne pas oublier de **bien valider** chaque fenêtre par le bouton « OK »





Pause questions

Chapitre 2



Premier programme

Premier code

Compilation

Exécution

Syntaxe basique

Afficher dans la console

Opérateurs arithmétiques

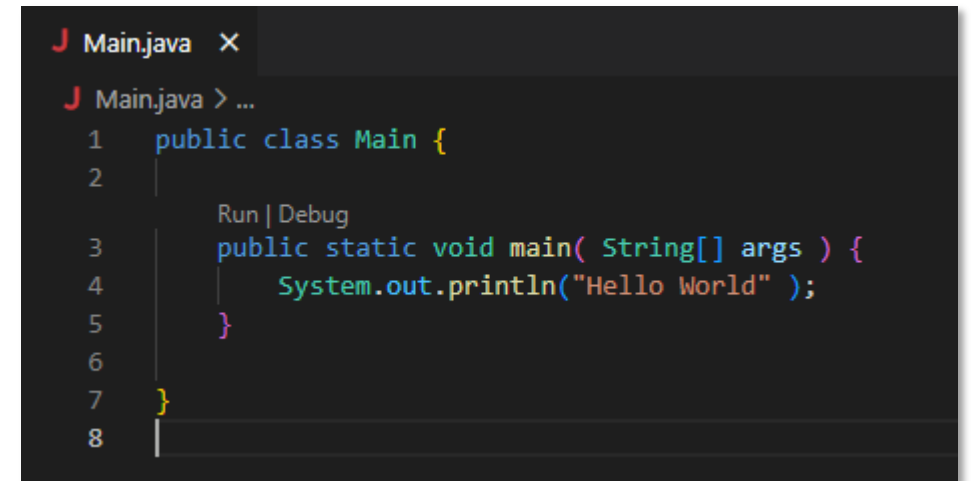
Les commentaires

Ecrire notre premier script

Pour créer une application java, nous devons commencer par créer son fichier principal : [Main.java](#).

Commençons ici avec un simple programme affichant « Hello world » dans la console. Nous en décrirons les instructions plus tard.

Une fois le code ci-contre ajouté à votre fichier, n'oubliez pas de sauvegarder.



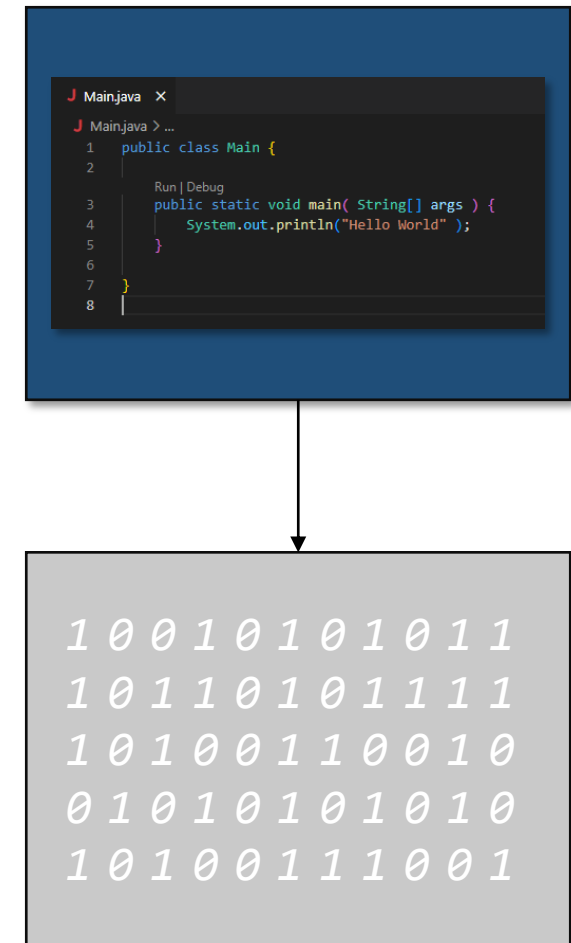
```
J Main.java x
J Main.java > ...
1 public class Main {
2
3     Run | Debug
4     public static void main( String[] args ) {
5         System.out.println("Hello World" );
6     }
7 }
8
```

Compilation

Avant de pouvoir exécuter notre programme, nous allons devoir passer par une étape de construction du programme à partir du code écrit.

Il s'agit de la compilation que nous opérerons avec la commande suivante :

```
javac Main.java
```

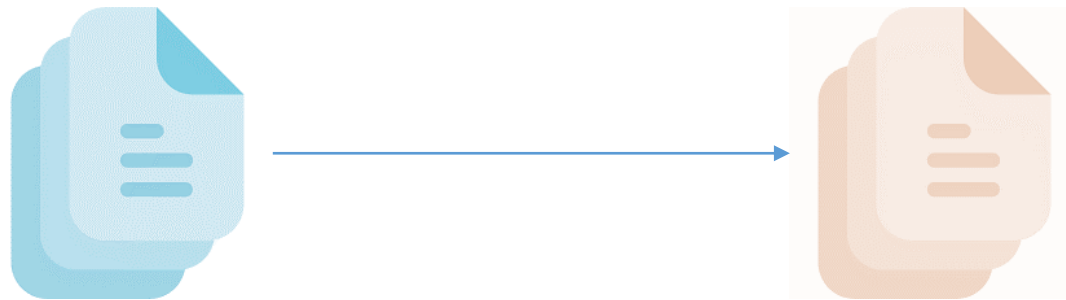


Compilation

Attention, lorsque nous aurons plusieurs fichiers dans notre application, il faudra compiler chacun d'entre eux.

Nous aurons alors à lister chaque fichier.java à la suite, comme par exemple :

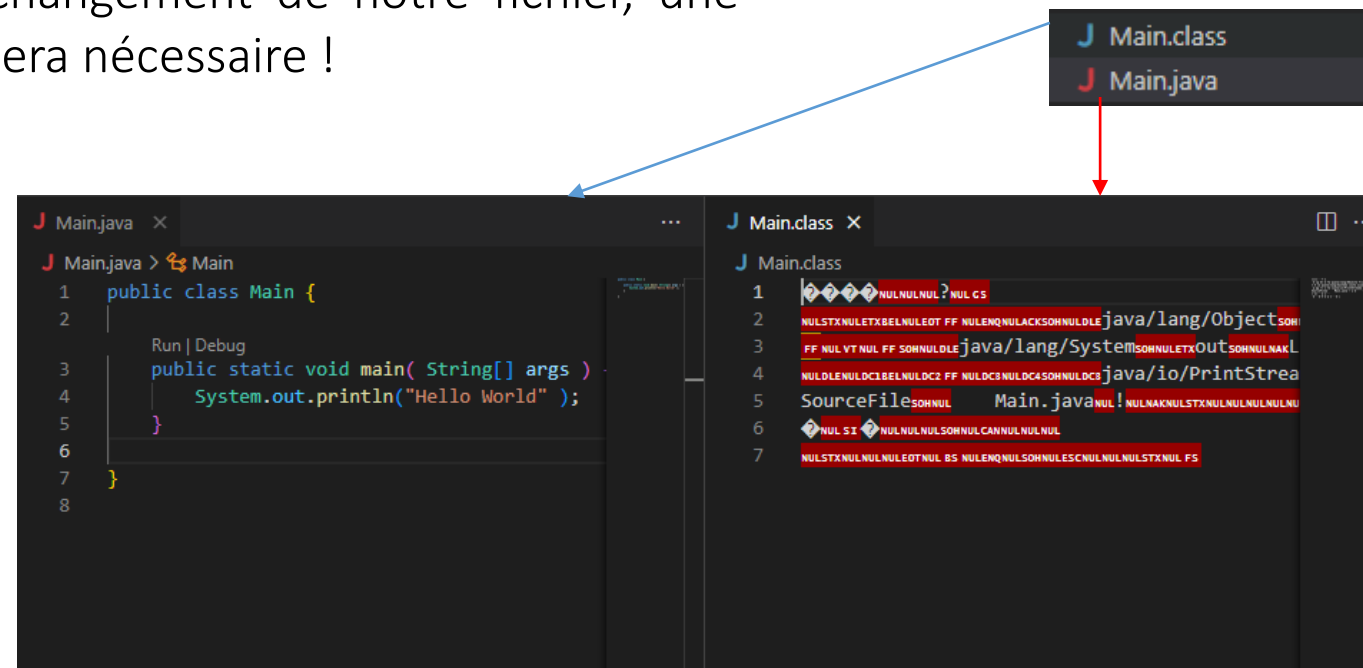
```
javac Main.java Database.java Network.java
```



Compilation

Une fois cette opération terminée, un nouveau fichier fait son apparition : [Main.class](#). Il s'agit de notre programme compilé.

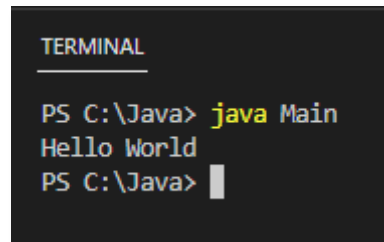
Attention, à chaque changement de notre fichier, une nouvelle compilation sera nécessaire !



Exécution

Il est maintenant possible d'exécuter notre programme en indiquant la commande suivante à votre terminal :


java Main

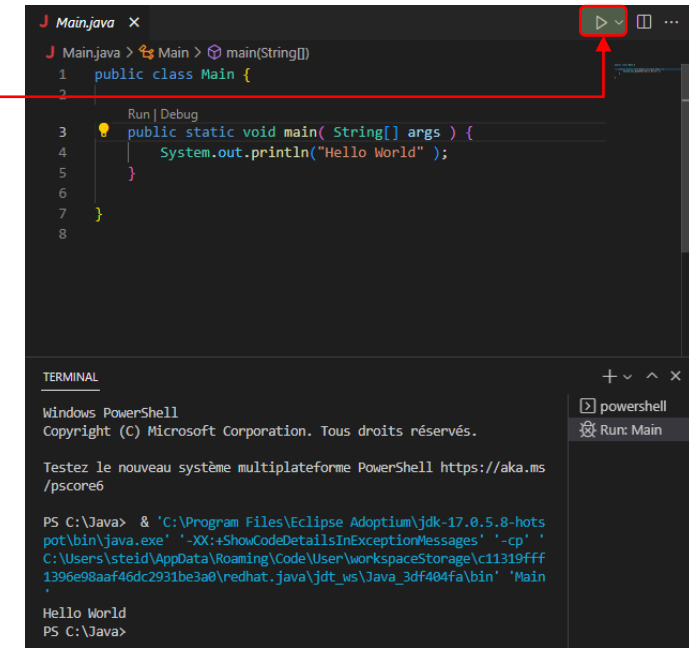
A screenshot of a terminal window with a dark background. The title bar at the top says 'TERMINAL'. The prompt 'PS C:\Java>' is followed by the command 'java Main' in yellow. The output 'Hello World' is displayed on the next line. The prompt 'PS C:\Java>' is followed by a white cursor on the third line.

```
TERMINAL
PS C:\Java> java Main
Hello World
PS C:\Java> 
```

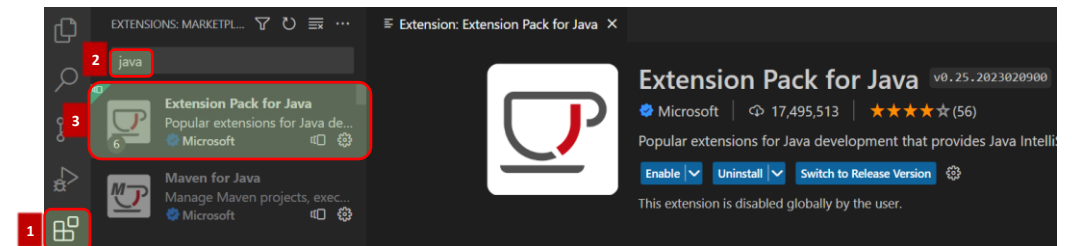
Notez bien l'absence d'extension ! Java ira de lui-même chercher le .class et ses dépendances.

Exécution

Une autre possibilité pour compiler et exécuter d'une traite votre programme est de cliquer sur l'icone  , en haut à droite de la fenêtre de VSCode.

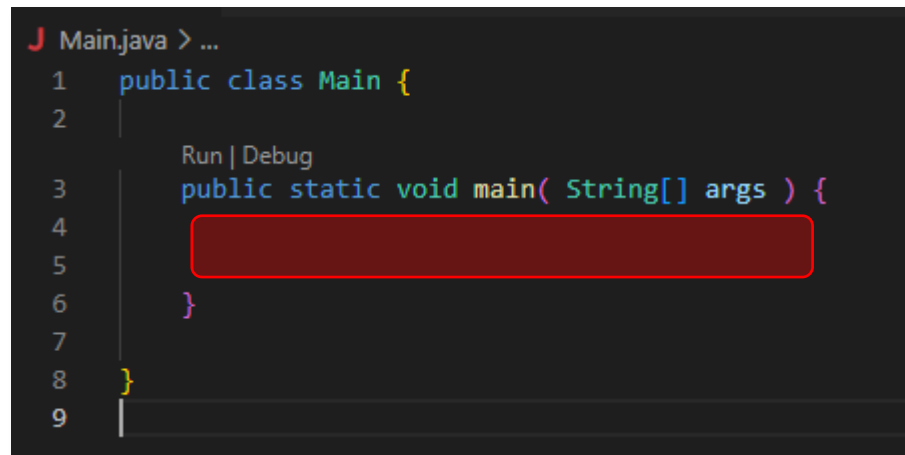


Pour obtenir cette icone et d'autres outils appréciables pour le développement java, il existe un module pour VSCode nommé Extension Pack for Java.

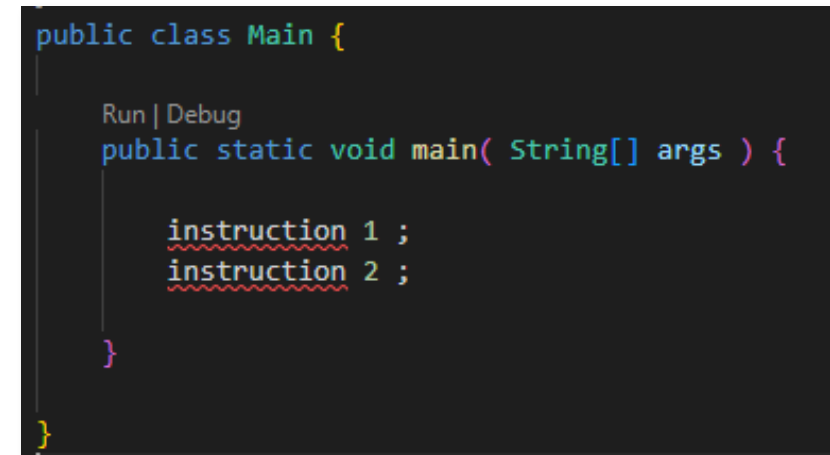


Syntaxe basique

Revenons sur la syntaxe évoquée plus tôt. Pour le moment, tout notre code devra se placer entre les accolades de **la partie rouge**.



```
1 public class Main {  
2  
3     Run | Debug  
4     public static void main( String[] args ) {  
5           
6     }  
7  
8 }  
9
```



```
public class Main {  
    Run | Debug  
    public static void main( String[] args ) {  
        instruction 1 ;  
        instruction 2 ;  
    }  
}
```

On y inscrira nos instructions qui seront exécutées de haut en bas dans l'ordre.

Chaque instruction devra se terminer par un « ; »

Afficher dans la console

Notre première instruction nous permettra d'afficher du texte dans la console.

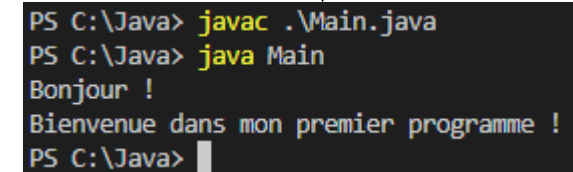
Il s'agira de la fonction :

```
System.out.println(“ Texte ”);
```

Le texte entre les guillemets sera alors affiché dans la console tel quel, suivi d'un retour à la ligne.



```
J Main.java > Main
1 public class Main {
2
3     Run | Debug
4     public static void main( String[] args ) {
5         System.out.println( "Bonjour !" );
6         System.out.println( "Bienvenue dans mon premier programme !" );
7     }
8
9 }
```

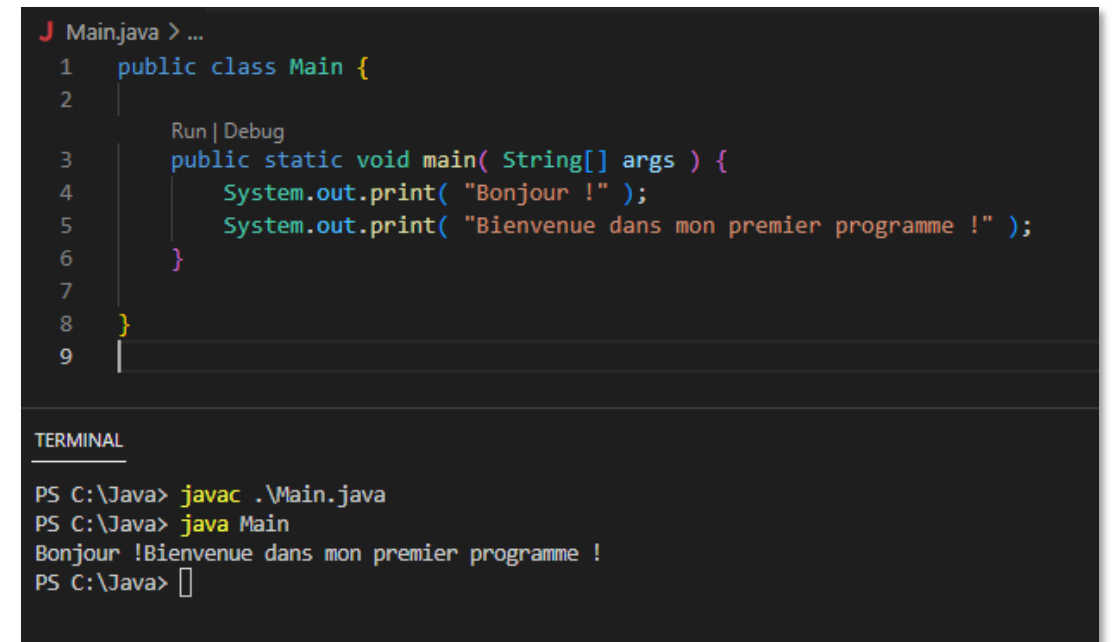


```
PS C:\Java> javac .\Main.java
PS C:\Java> java Main
Bonjour !
Bienvenue dans mon premier programme !
PS C:\Java> 
```


Afficher dans la console

Notez que si l'on remplace la fonction `println` par `print`, nous n'aurons alors pas de retour à la ligne automatique en fin d'affichage.

Le rendu est alors sensiblement différent.



```
J Main.java > ...
1 public class Main {
2
3     Run | Debug
4     public static void main( String[] args ) {
5         System.out.print( "Bonjour !" );
6         System.out.print( "Bienvenue dans mon premier programme !" );
7     }
8 }
9

TERMINAL
PS C:\Java> javac .\Main.java
PS C:\Java> java Main
Bonjour !Bienvenue dans mon premier programme !
PS C:\Java> 
```

Afficher dans la console

Lorsque l'on souhaite afficher une suite de caractères, on encadre donc cette suite de guillemets.

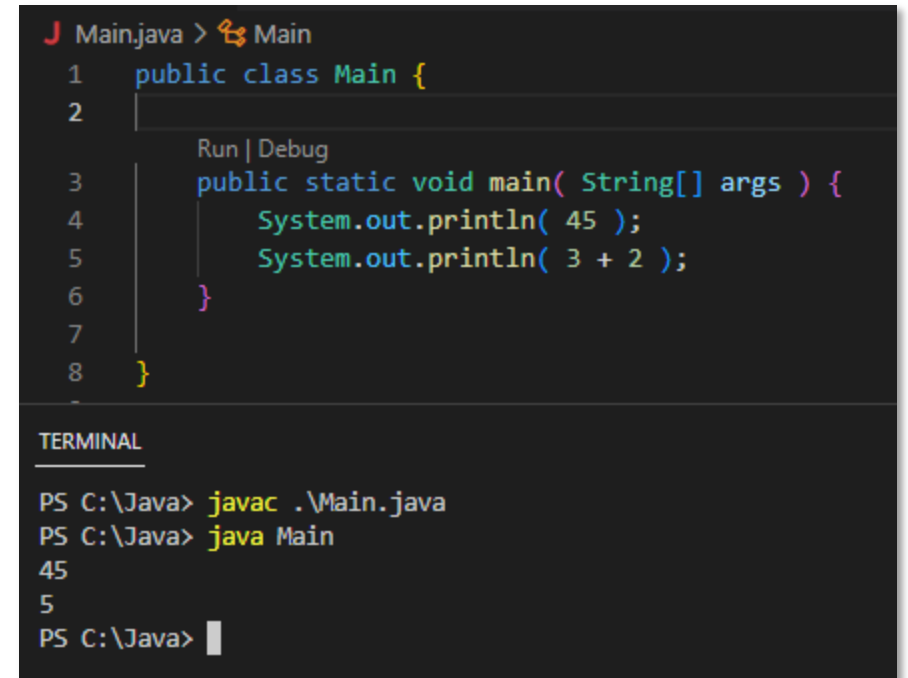
Certains caractères particuliers demanderont une syntaxe particulière.

Caractère	Description
\n	Retour à la ligne
\t	Tabulation
\"	Guillemet

Afficher dans la console

Il est aussi possible d'afficher une valeur numérique, c'est-à-dire un nombre ou le résultat d'un calcul numérique.

Dans ce cas, pas besoins de guillemets.



The screenshot shows an IDE window titled 'Main.java > Main'. The code editor contains the following Java code:

```
1 public class Main {  
2  
3     Run | Debug  
4     public static void main( String[] args ) {  
5         System.out.println( 45 );  
6         System.out.println( 3 + 2 );  
7     }  
8 }
```

Below the code editor is a 'TERMINAL' window showing the execution of the code:

```
PS C:\Java> javac .\Main.java  
PS C:\Java> java Main  
45  
5  
PS C:\Java> 
```

Opérateur arithmétiques

Notez que dans le cadre de l'addition précédente (3 + 2) nous avons utilisé l'opérateur arithmétique « + ».

Il existe d'autres opérateurs arithmétique en Java qui peuvent être utilisés de la même façon.

Opérateur	Description	Exemple	Résultat
+	Addition	2 + 2	4
-	Soustraction	2 - 2	0
*	Multipliation	2 * 3	6
/	Division	6 / 2	3
%	Modulo (Reste de division)	7 % 2	1

Les commentaires

En programmation, il est souvent utile d'annoter son code sans influencer sur son exécution.

Pour ce faire, deux méthodes :

- Tout ce qui se trouvera après et sur la même ligne qu'un « `//` » sera un commentaire
- Tout ce qui se trouvera entre « `/*` » et « `*/` » sera un commentaire.

```
6      // Je suis un commentaire
7      System.out.println("Je ne suis pas un commentaire");
8
9      /*
10     Tout ce code est un commentaire, même l'instruction
11     ci-dessous (elle ne sera donc pas exécutée)
12     System.out.println("Hello world !");
13     */
```

TERMINAL

```
PS C:\Java> javac .\Main.java
PS C:\Java> java Main
Je ne suis pas un commentaire
PS C:\Java> 
```



Pause questions

Chapitre 3



Variables

Types de valeur

Qu'est-ce qu'une variable ?

Créer une variable

Affectation et réaffectation

Lecture de la variable

Opérateurs de réaffectation

Constantes

Types de valeur

En Java, il existe plusieurs types de valeurs. Nous en avons vu 2 précédemment : les chaînes de caractères et les nombres entier.

Voici une liste de quelques uns des types les plus communément utilisé en Java.

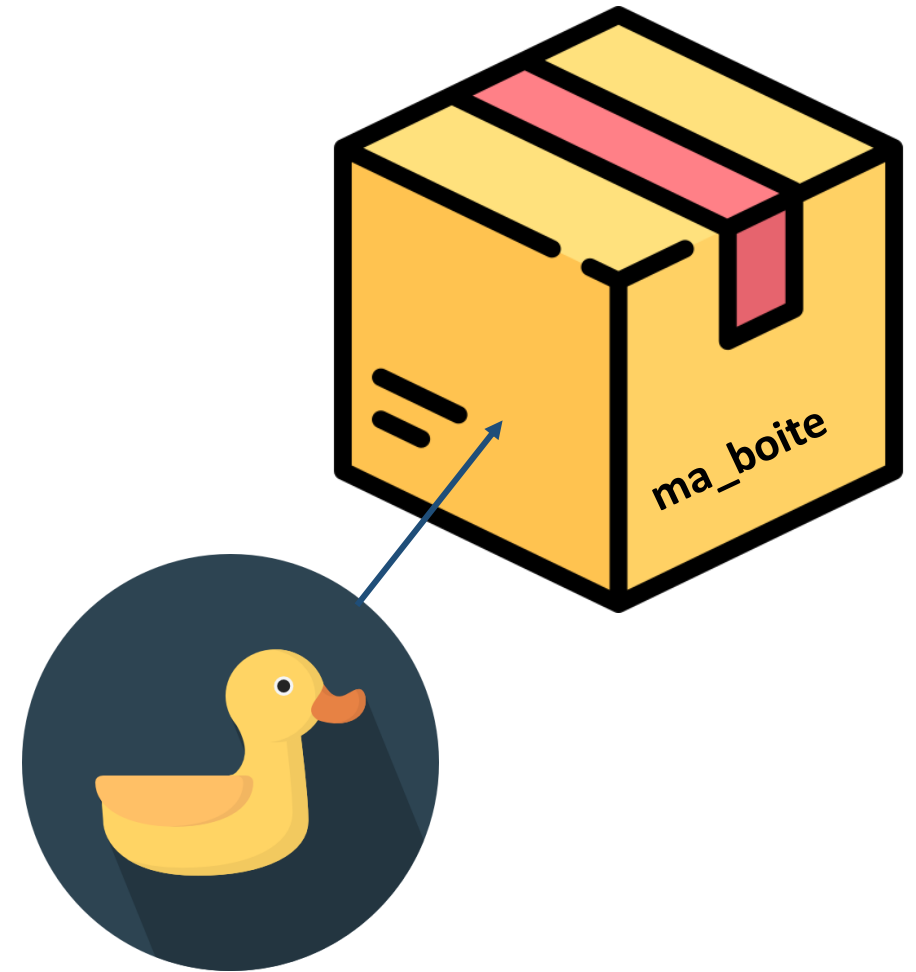
Type	Description	Exemple
Int	Nombre entier	15, -10, 124
Float	Nombre décimal	7.52, 0.652, -142.1
Char	Caractère	A, R, !, W
Boolean	Booléen	True, False
String	Chaine de caractères	« Hello », « Good morning »

Qu'est ce qu'une variable ?

Les variables sont des cases mémoires nommées dans lesquelles nous pouvons stocker une valeur.

Par exemple, ci-contre, on imagine que notre variable est une boîte portant le nom « `ma_boite` » et dont le contenu est un canard.

Plus tard, le contenu de cette boîte pourra changer, d'où le nom de variable.



Créer une variable

Une variable en Java doit être typée. C'est-à-dire que l'on va réserver un espace en mémoire pour y stocker un type d'information bien précis.

Lors de la création, on indique donc d'abord le type de la variable puis on lui donne un nom.

Attention, le nom d'une variable ne peut contenir que des caractères alphanumériques et/ou le « _ » et ne doit pas commencer par un chiffre !

```
int variable_entiere ;  
boolean variable_booleene ;  
String variable_chaine ;
```

Affectation et réaffectation

Pour attribuer une valeur à notre variable, on utilisera l'opérateur « = ». On appelle cette opération une affectation.

On placera la variable de destination à gauche et la valeur à droite de l'opérateur d'affectation. Il est possible d'effectuer autant d'affectation, et donc de réaffectation, que l'on souhaite.

Enfin, il est possible, et même conseillé, d'affecter une valeur à notre variable au moment de sa création.

```
int ma_variable = 10 ;  
ma_variable = 15 ;  
ma_variable = 10 + 7 ;
```

Lecture de la variable

Pour utiliser la valeur stockée dans notre variable, il suffit de l'invoquer par son nom. Notre variable sera alors en quelque sorte remplacée par sa valeur actuelle.

On peut ainsi effectuer un affichage de celle-ci, des calculs ou diverses autres opérations basées sur sa valeur actuelle.

```
6      int ma_variable = 10 ;
7      System.out.println( ma_variable );
8      ma_variable = ma_variable * 2 ;
9      System.out.println( ma_variable );
10
```

TERMINAL

```
PS C:\Java> javac .\Main.java
PS C:\Java> java Main
10
20
PS C:\Java> 
```

Ici en ligne 8, on réaffecte à notre variable sa propre valeur actuelle multipliée par 2

Opérateurs de réaffectation

Pour ajouter une valeur particulière à notre variable, il est possible d'en réaffecter la valeur à partir de sa valeur actuelle comme vu précédemment.

Cependant, l'opérateur `+=` simplifiera grandement notre syntaxe en ajoutant directement la valeur à sa droite à la valeur actuelle de la variable à sa gauche.

```
int ma_variable = 10 ;           // ma_variable vaut 10
ma_variable = ma_variable + 5 ;  // ma_variable vaut 15
ma_variable += 3 ;               // ma_variable vaut 18
```

Opérateurs de réaffectation

On peut aussi incrémenter très simplement la valeur d'une variable de 1 avec l'opérateur « ++ » après celui-ci.

```
3  ✓ public static void main( String[] args ) {  
4  
5      int ma_variable = 10 ;           // ma_variable vaut 10  
6      ma_variable++ ;                 // ma_variable vaut 11  
7      System.out.println( ma_variable );  
8  
9  }
```

TERMINAL

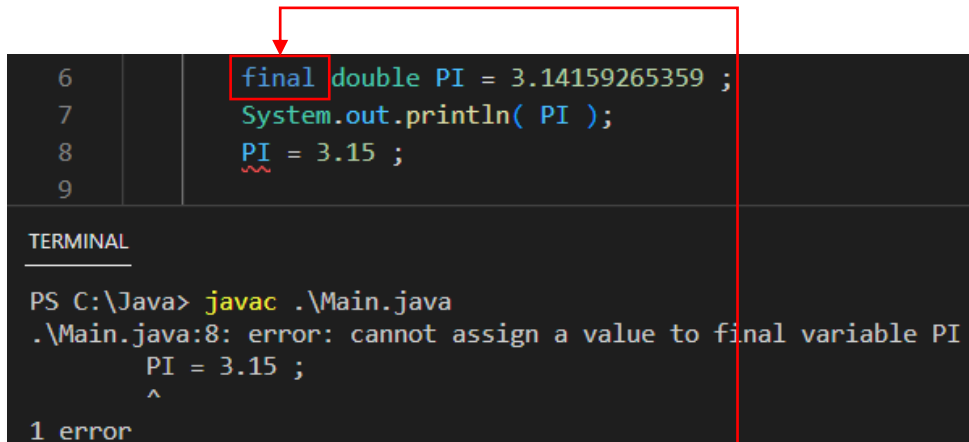
```
PS C:\Java> javac .\Main.java  
PS C:\Java> java Main  
11  
PS C:\Java> 
```

Opérateurs de réaffectation

Il existe d'autres opérateur de réaffectation dont voici la liste. Ils s'utilise de la même manière que vu précédemment.

Opérateur	Signification	Exemple (partant de x = 5)	X
+=	Incrémentation	x += 2	7
-=	Décrémentation	x -= 2	3
*=	Multiplication	x *= 2	10
/=	Division	x /= 2	2,5
%=	Modulo	x %= 2	1
++	Incrémentation (de 1)	x++	6
--	Décrémentation (de 1)	x--	4

Constantes



```
6 final double PI = 3.14159265359 ;
7 System.out.println( PI );
8 PI = 3.15 ;
9
```

TERMINAL

```
PS C:\Java> javac .\Main.java
.\Main.java:8: error: cannot assign a value to final variable PI
    PI = 3.15 ;
    ^
1 error
```

Les variables constantes sont des variables classique dont on ne peut pas modifier la valeur attribuée lors de sa création.

C'est un peu perturbant de rendre ainsi une variable invariable. C'est pourtant bien utile pour définir une sorte d'alias à une valeur, comme PI par exemple.

Pour définir une variable comme constante, on en précédera la définition avec le mot clef « **final** ».

Caster

Que ce soit une valeur brute ou une variable, il est possible de la **caster** en un autre type compatible.

Caster signifie que l'on va transformer le type de la valeur pour la lire sous un autre angle. Par exemple, pour transformer un entier en décimal, ou encore un entier en caractère.

Pour **caster** une valeur dans un type particulier, il suffira de précéder cette valeur (ou variable) par le nouveau type entre `()`.

```
System.out.println( 64 );  
System.out.println( (float) 64 );  
System.out.println( (char) 64 );
```



Pause questions

Chapitre 4



Programmes dynamiques

Opération sur les chaînes

Imports simples

Entrée clavier

Segments d'instructions

Portée des variables

Opérations sur les chaines - *Formatage*

```
5 String chaine_1 = "Hello" ;
6 String chaine_2 = "world" ;
7
8 System.out.println(
9     chaine_1 + " " + chaine_2 + " !"
10 );
11
```

TERMINAL

Hello world !

Il sera très souvent pratique de pouvoir afficher la valeur de différentes entités dans le terminal, et donc de les intégrer à nos chaines de caractères.

```
5 String nom = "Jean" ;
6 int age = 34 ;
7
8 System.out.println(
9     nom + " a " + age + " ans"
10 );
11
```

TERMINAL

Jean a 34 ans

Il par exemple possible de **fusionner plusieurs chaines entre elles**, ou encore de leur fusionner d'autres types, comme des entiers.

On utilisera l'opérateur « + » pour ce faire. On n'appelle cette opération une **concaténation**.

Opérations sur les chaînes - *Formatage*

Pour améliorer la lisibilité du code et faciliter certaines manipulations, il est également possible de se servir de la méthode `String.format()`.

Cette méthode prendra en premier paramètre une chaîne à formater, dans laquelle nous placerons des symboles à remplacer par certains types de valeurs.

Par exemple, `%s` sera remplacé par une autre chaîne, `%d` par un entier et `%f` par un flottant. L'ensemble des valeurs correspondantes seront fournies dans l'ordre à la suite de la chaîne.

```
5      String nom = "Jean" ;
6      int age = 34 ;
7      double bank = 452.68 ;
8
9      System.out.println(
10         String.format(
11             format: "%s a %d ans et dispose de %fE sur son compte en banque."
12             nom, age, bank
13         )
14     );
15
```

TERMINAL

Jean a 34 ans et dispose de 452,680000E sur son compte en banque.

Opérations sur les chaines - *Formatage*

Voici un tableau des différents symboles permettant une intégration de différents type de données sous différentes formes au sein d'une chaine formatée.

Les lignes colorée sont les formats les plus utiles. En bleu, ceux cité précédemment, en vert les nouveaux.

Symbole	Type de données	Affichage
%a	Flottant	Affichage hexadécimal d'un flottant
%b	Tout type	Affiche « true » si valeur non nulle, « false » sinon
%c	Caractère	Affiche le caractère
%d	Entier	Affiche l'entier
%e	Flottant	Affiche le nombre décimal avec une notation scientifique
%f	Flottant	Affiche le nombre décimal
%g	Flottant	Affiche le nombre décimal avec une notation scientifique si la précision le nécessite
%h	Tout type	Format hexadécimal de la valeur hachée (par <code>hashCode()</code>)
%n	---	Affiche un séparateur (dépend du système)
%o	Entier	Affiche l'entier au format octal
%s	Tout type	Affiche la valeur sous forme de chaine de caractère (via <code>toString()</code>)
%x	Entier	Affichage de l'entier en hexadécimal

Opérations sur les chaînes - *Formatage*

Enfin, concernant l'affichage des flottant avec %f, il est possible de définir le nombre de caractère à intégrer avant et après la virgule.

Par exemple, le symbole %**5**.**4**f affichera la **partie entière** avec 5 caractères et la **partie décimale** avec 4.

```
5 double x = 10.129456789 ;
6
7 System.out.println(
8     String.format("X vaut %.2f", x)
9 );
10
11 System.out.println(
12     String.format("X vaut %10.2f", x)
13 );
14
15 System.out.println(
16     String.format("X vaut %.15f", x)
17 );
18
```

TERMINAL

```
X vaut 10,13
X vaut      10,13
X vaut 10,129456789000000
```

Les caractères superflus de la partie entières seront des espaces, ceux de la partie décimale des 0.

En cas de perte de précision de la partie décimale, on effectuera un arrondi.

Opérations sur les chaînes - *Taille*

La fonction `length()` permet d'obtenir un entier représentant le nombre de caractères contenus dans la chaîne.

```
7 String ma_chaine = "Ceci est une longue chaîne de caractère" ;  
8 System.out.println( ma_chaine.length() );  
9  
TERMINAL  
39
```


Opérations sur les chaines - *Taille*

La fonction `length()` permet d'obtenir un entier représentant le nombre de caractères contenus dans la chaine.

```
7 String ma_chaine = "Ceci est une longue chaine de caractère" ;  
8 System.out.println( ma_chaine.length() );  
9  
TERMINAL  
39
```

```
7 String ma_chaine = "" ;  
8 System.out.println( ma_chaine.isEmpty() );  
9  
TERMINAL  
true
```

La fonction `isEmpty()` retourne quant à elle un booléen, vrai si la chaine est vide, faux sinon.

Opérations sur les chaines – *N° de caractère*

La fonction `charAt(<i>)` permet d'obtenir le caractère à l'index `<i>`. Notez qu'en le castant en entier, nous pourrions récupérer le n° du caractère.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	.	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	:	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

```
7      String ma_chaine = "Hello world !" ;
8      System.out.println(
9          String.format(
10             "Le caractère n°4 de la chaine '%s' est le n°d",
11             ma_chaine.charAt(4),
12             (int) ma_chaine.charAt(4)
13         )
14     );
```

TERMINAL

Le caractère n°4 de la chaine 'o' est le n°111

Vous pourrez d'ailleurs le vérifier sur n'importe quelle table ASCII comme celle-ci contre.

Opérations sur les chaines – *Egalité*

L'opérateur `==` ne sera pas toujours capable de vérifier une égalité entre deux chaines de contenu identique.

Pour réaliser cette opération, nous préférons utiliser la fonction `equals()` qui s'utilise sur et avec une chaine.

```
8      String chaine_1 = "hello" ;
9      String chaine_2 = "hello" ;
10
11      System.out.println( chaine_1.equals( chaine_2 ) );
12
```

TERMINAL

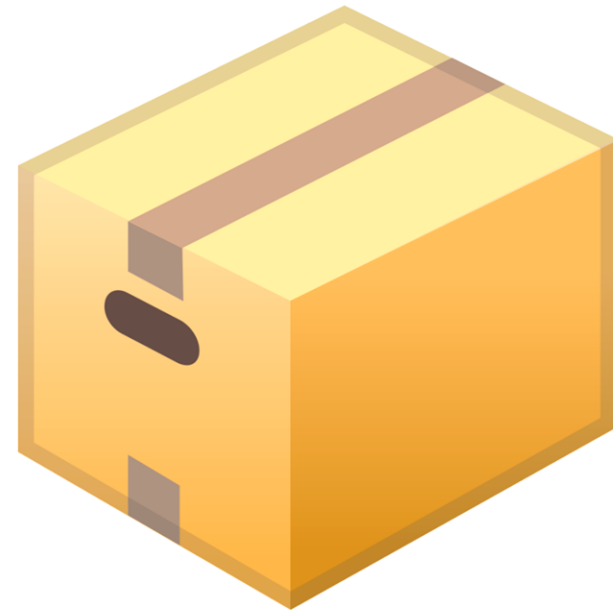
true

Imports simples

En Java, et comme dans beaucoup d'autres langages, il est fréquent d'utiliser ou réutiliser des fonctionnalités externes à notre fichier actuel, voire à notre programme.

On appelle ces sources de fonctions externes des packages. Nous les verrons plus en détails plus tard.

Pour l'heure, intéressons nous simplement à l'import de fonctionnalités depuis des packages présents dans notre installation native de Java.



Imports simples

Pour importer une fonctionnalité précise d'un package, nous allons utiliser le mot clef `import`.

Celui-ci devra être suivi du nom du package dans lequel se trouve notre fonctionnalité.

Le « `.` » nous permettra ensuite d'aller retrouver notre fonctionnalité. Un exemple ci-contre avec l'import de la fonctionnalité « `MyTool` » depuis le package « `tools` ».

Package

```
import tools.MyTool ;
```

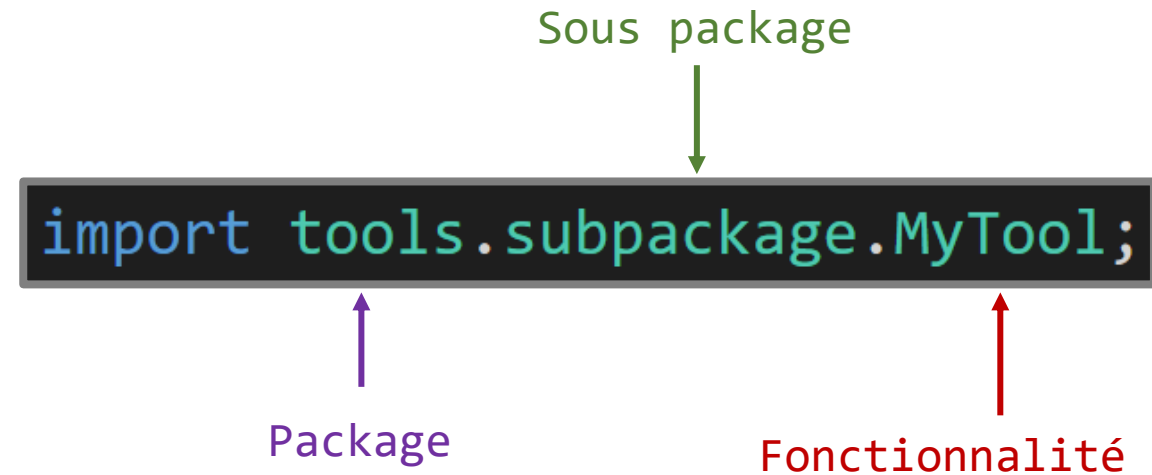
Fonctionnalité

Imports simples

Il n'est également pas rare d'avoir des packages décomposés en sous packages.

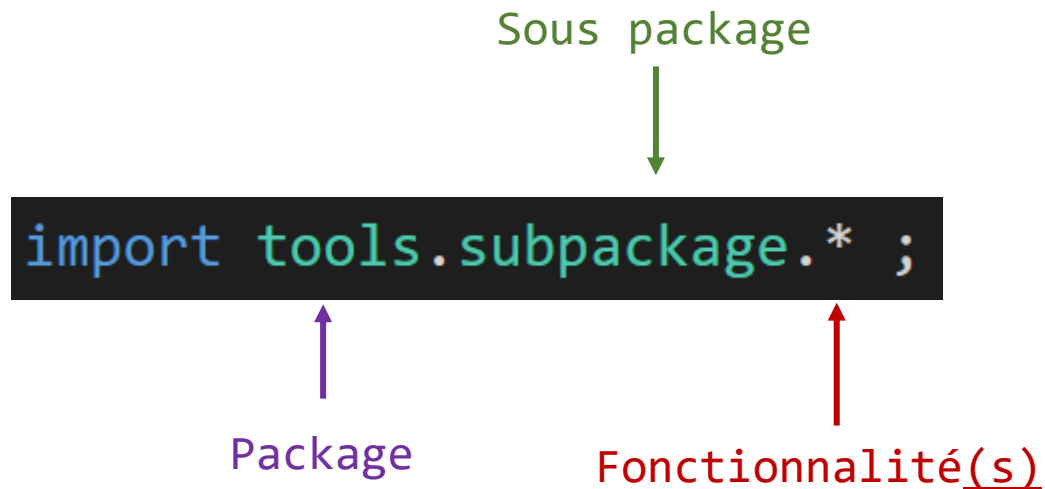
Dans ce cas, il faudra accéder dans l'ordre à tous les sous-packages avant de parvenir à notre fonctionnalité. Là encore, c'est le « . » qui nous le permettra.

Ci-contre, nous importons « **MyTool** » depuis el sous-package « **subpackage** », lui-même dans le package « **tools** ».



Imports simples

Enfin, pour importer toutes les fonctionnalités d'un même paquet, il suffit de remplacer le nom de la fonctionnalité par « * ».



Entrée clavier

Il est possible de demander à un utilisateur de notre programme terminal d'entrer une valeur au clavier. Pour ce faire, il nous faut d'abord importer `java.util.Scanner`.

`Scanner` est un Objet disposant de différentes fonctionnalités pour l'obtention de saisies dans le terminal.

Dans un premier temps, il va nous falloir créer un Scanner avec le code ci-contre.

```
Scanner scan = new Scanner( System.in );
```


Entrée clavier

Nous verrons plus en détails le pourquoi de cette syntaxe plus tard, mais dans les grandes lignes :

```
Scanner scan
```

On crée une nouvelle variable pouvant accueillir notre Scanner et la nommons « scan ».

```
= new Scanner
```

Nous affectons à cette variable un nouveau scanner...

```
( System.in );
```

...qui a besoins, pour se créer et fonctionner, d'un paramètre spécifique. Ici **System.in** indique la source de la capture d'information (le terminal).

Entrée clavier

A partir de là, il nous est possible d'accéder à différentes fonctions depuis notre variable contenant le scanner via le « . ».

Par exemple, la fonction `nextLine()` de `Scanner` nous permettra d'obtenir une suite de caractères entrée par l'utilisateur jusqu'à ce qu'il entre un retour à la ligne.

Il existe aussi `nextInt()` qui cette fois retournera un entier entré au clavier.

```
Scanner scan = new Scanner( System.in );

System.out.println(x: "Entrez votre nom :");
String name = scan.nextLine();

System.out.println(x: "Entrez votre age :");
int age = scan.nextInt();

System.out.println(
    String.format(
        format: "Bonjour %s, dans %d vous aurez un siècle !",
        name, 100 - age
    )
);
```

Entrée clavier

Pour terminer, il est recommandé d'utiliser la fonction « `close()` » de notre scanner pour le fermer et indiquer sa fin de mission.

```
Scanner scan = new Scanner( System.in );

System.out.println(x: "Entrez votre nom :");
String name = scan.nextLine();

System.out.println(x: "Entrez votre age :");
int age = scan.nextInt();

System.out.println(
    String.format(
        format: "Bonjour %s, dans %d vous aurez un siècle !",
        name, 100 - age
    )
);

scan.close();
```



```
Entrez votre nom :
Arnaud
Entrez votre age :
42
Bonjour Arnaud, dans 58 vous aurez un siecle !
```

Segments d'instructions

Dans certains cas que nous verrons très prochainement, il sera important de pouvoir délimiter des zones dans notre code.

Il s'agira simplement de suites d'instructions encadrées par des `{ }` (*accolades*) et que l'on pourra utiliser, réutiliser, conditionner, etc...

```
{  
    System.out.println("Ceci est un segment de code");  
    System.out.println("Il comporte 2 instructions");  
}  
  
{  
    System.out.println("En voici un second avec une unique instruction");  
}
```

Segments d'instructions

On pourra également retrouver imbriquer ces segments les uns dans les autres de diverses manières.

```
{  
    System.out.println("Ceci est un segment de code");  
    {  
        System.out.println("Un segment dans un segment ?!");  
    }  
    System.out.println("Il comporte 2 instructions");  
}  
  
{  
    System.out.println("En voici un second avec une unique instruction...");  
    {  
        System.out.println("... et un sous-segment");  
        System.out.println("en comportant 2");  
    }  
}
```

Segments d'instructions

Remarquez au passage que le « main » dans lequel nous programmons depuis le début est délimité lui aussi par ces accolades.

```
public static void main( String[] args ) {  
    // Ceci est le programme principal  
}
```

Il s'agit là aussi d'un segment de code : le segment du programme principal.

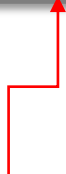
Portée des variables

Ce que l'on appelle la « **portée** » des variables est tout simplement leur **champ d'action**. En d'autres termes, une variable ne sera utilisable que dans un certain cadre.

Ce cadre d'utilisation est le **segment d'instructions** dans lequel elle a été créée.

Par extension, la variable sera **valable** dans tous les **sous-segments** qui comprennent le segment de code dans lequel elle a été défini.

```
7      int x ;
8
9      {
10         int y ;
11         System.out.println( x );
12         System.out.println( y );
13     }
14
15     System.out.println( x );
16     System.out.println( y );
```



```
Main.java:16: error: cannot find symbol
    System.out.println( y );
                       ^
    symbol:   variable y
    location: class Main
1 error
error: compilation failed
```

Portée des variables

On dit alors d'une variable définie **dans le segment actuel** qu'elle en est **locale**, et d'une variable définie **dans un segment parent** qu'elle en est **globale**.

Ci-contre, **X** est donc globale au segment lignes 9 à 13 mais local au programme dans lequel il se trouve.

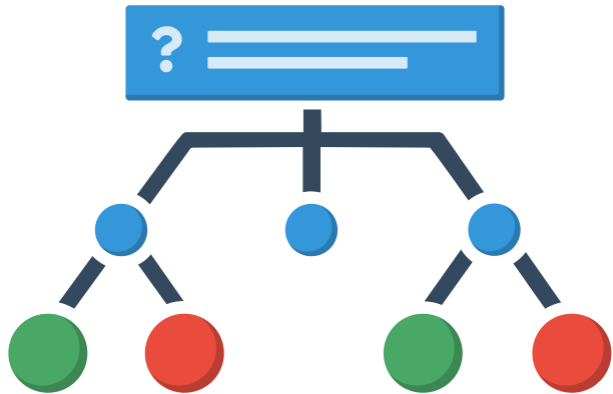
Y quant à elle est une variable locale au segment lignes 9 à 13 et ne peut donc pas être utilisée en dehors, d'où l'erreur ligne 16.

```
7      int x ;
8
9      {
10         int y ;
11         System.out.println( x );
12         System.out.println( y );
13     }
14
15     System.out.println( x );
16     System.out.println( y );
```




Pause questions

Chapitre 5



Structures conditionnelles

Conditions

Combinaisons logiques

Structure IF

Structure Switch

Opérateur ternaire

Conditions

Les conditions en programmations sont des expressions ayant une valeur booléenne, c'est-à-dire vrai ou fausse.

Il s'agira souvent de comparaisons, d'égalité, d'inégalité, de supériorité etc...

On utilisera des opérateurs conditionnels pour construire ces conditions, comme un calcul arithmétique. Ci-contre quelques exemples.

```
5      int ma_variable = 10 ; // ma_variable vaut 10
6      ma_variable++ ;       // ma_variable vaut 11
7      System.out.println( ma_variable == 11 );
8      System.out.println( ma_variable == 10 );
9      System.out.println( ma_variable > 5 );
10
```

TERMINAL

```
PS C:\Java> javac .\Main.java
PS C:\Java> java Main
true
false
true
```

Conditions

Voici un tableau regroupant les différents opérateurs conditionnels, leur signification et un exemple d'utilisation.

Opérateur	Signification	Exemple	Valeur
==	Egalité	41 == 62	False
!=	Inégalité	41 != 62	True
>	Strictement supérieur	10 > 5	True
<	Strictement inférieur	10 < 5	False
>=	Supérieur ou égal	11 >= 5 + 6	True
<=	Inférieur ou égal	11 <= 5 - 3	True

Combinaisons logiques

Il est possible de combiner ces expressions logiques sur la base de « et », de « ou » et d'inversions logiques.

On utilisera cette fois des opérateurs de combinaison logique. On retrouvera à sa droite et à sa gauche les conditions vues précédemment.

Ci-contre, les 3 opérateurs de combinaison logique.

Opérateur	Signification	Exemple	Valeur
&&	Et	10 == 10 && 5 > 6	False
	Ou	10 == 10 5 > 6	True
!	Inverse	!10 == 10	False

```
8      System.out.println( 10 == 10 && 5 > 6 );
9      System.out.println( 10 == 10 || 5 > 6 );
10     System.out.println( !(10 == 10) );
11
```

TERMINAL

```
PS C:\Java> javac .\Main.java
PS C:\Java> java Main
true
false
true
```

Combinaisons logiques

Voici les tables de vérité de ces 3 opérateurs :

Opérateur && (Et) – A && B		
A \ B	True	False
True	True	False
False	False	False

Opérateur (Ou) – A B		
A \ B	True	False
True	True	True
False	True	False

Opérateur ! (Not) – !A		
A	True	False
	False	True

Structure IF

Lors de l'exécution d'un programme, ou ici d'un script, il est fréquent de devoir faire des choix quant aux instructions à exécuter.



Ainsi, un ensemble d'instruction pourra être exécuté plutôt qu'un autre en fonction de conditions particulières.

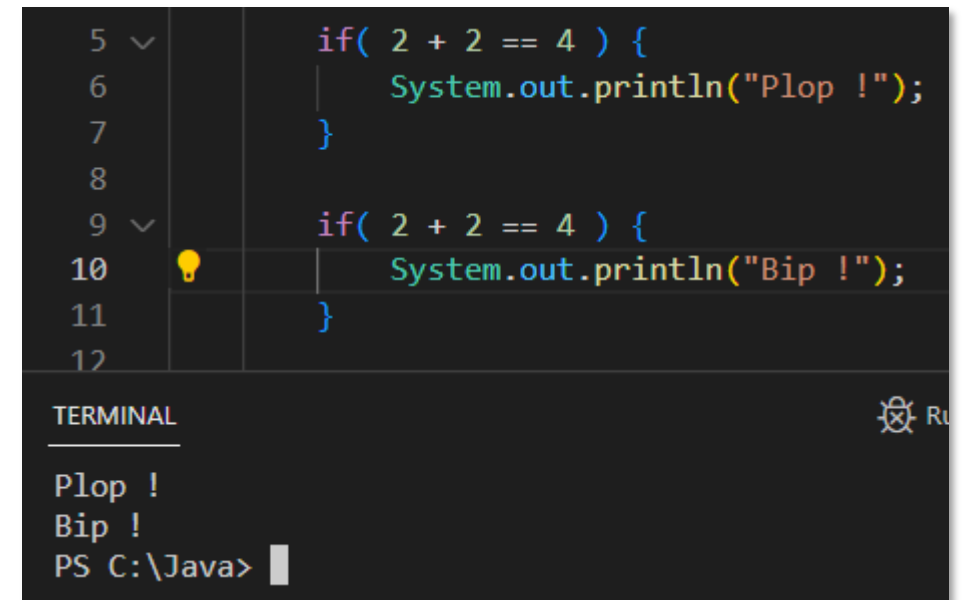
Structure IF – *Le IF*

On appelle ça une structure conditionnelle. Il en existe plusieurs en Javascript, dont la première est le if.

Il s'articule sous la forme :

```
if( condition ) {  
    instructions  
}
```

Les **instructions** ne s'exécuteront que si la **condition** est vraie. Tout ce qui est en dehors de la structure IF sera exécuté normalement.



```
5  ✓ if( 2 + 2 == 4 ) {  
6      System.out.println("Plop !");  
7  }  
8  
9  ✓ if( 2 + 2 == 4 ) {  
10     System.out.println("Bip !");  
11 }  
12
```

TERMINAL

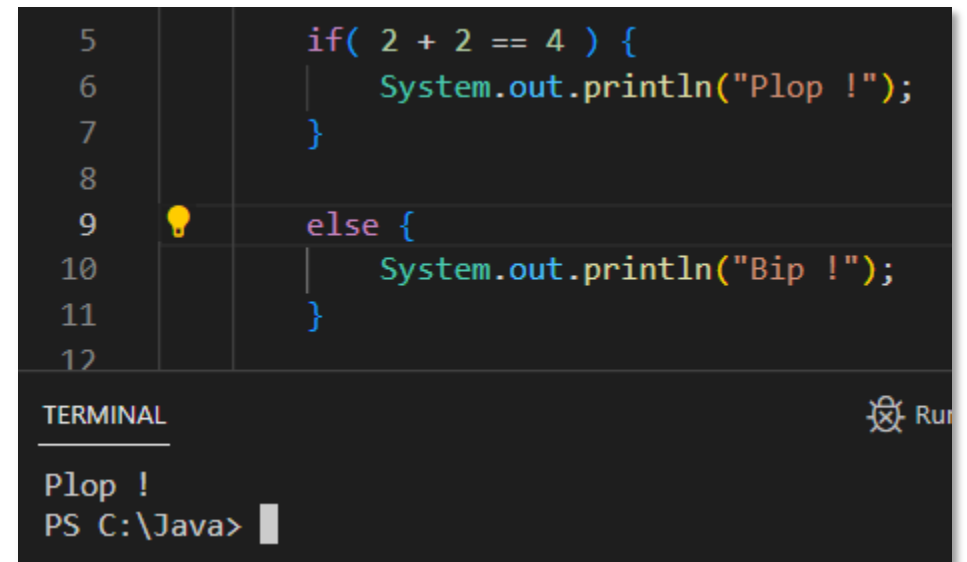
```
Plop !  
Bip !  
PS C:\Java>
```


Structure IF – *Le else*

On peut également ajouter un scénario en cas d'échec avec le mot clef **else** :

```
...  
else {  
    instructions si faux  
}
```

Les **instructions si faux** ne s'exécuteront que si la **condition** est fausse.



```
5      if( 2 + 2 == 4 ) {  
6          System.out.println("Plop !");  
7      }  
8  
9      else {  
10         System.out.println("Bip !");  
11     }  
12
```

TERMINAL

```
Plop !  
PS C:\Java>
```

The screenshot shows a code editor with a dark theme. Lines 5-12 of code are visible. Line 9 has a yellow lightbulb icon. Below the code, a terminal window shows the output 'Plop !' and the command prompt 'PS C:\Java>'. A 'Run' button is visible in the top right of the terminal area.


Structure IF – Le *else if*

Il est également possible d'ajouter des scénarios alternatifs sous la forme

```
...  
else if( condition_alternative ) {  
    instructions si vrai  
}  
...
```

Les `instructions si vrai` ne s'exécuteront que si la `condition_alternative` est vraie.

```
5      if( 2 + 2 == 7 ) {  
6          System.out.println("Plop !");  
7      }  
8      else if( 2 + 2 == 4 ) {  
9          System.out.println("Glup !");  
10     }  
11     else {  
12         System.out.println("Bip !");  
13     }  
14
```

TERMINAL  Run

```
Glup !  
PS C:\Java> 
```

Structure IF – *Le else if*

Autant le **if** et le **else** sont des instructions uniques (un seul par structure conditionnelle), autant il possible d'avoir une infinité de **else if**.

Chaque **else if** possède sa propre **condition** et ses propres **instructions**.



```
5      int age = 40 ;
6
7  ✓    if( age == 20 ) {
8        System.out.println("Yo !");
9      }
10  ✓   else if( age == 30 ) {
11        System.out.println("Hey !");
12      }
13  ✓   else if( age == 40 ) {
14        System.out.println("Bonjour !");
15      }
16  ✓   else if( age == 50 ) {
17        System.out.println("Salutation !");
18      }
19  ✓   else {
20        System.out.println("Grrr !");
21      }
22
```

TERMINAL Run: Main

```
Bonjour !
PS C:\Java>
```

Structure Switch

La structure `switch` est une autre structure conditionnelle assez différente.

Elle prend en paramètre une variable qu'elle comparera avec différentes valeurs.

Chaque valeur devra être présentée avec le mot clef `case`. **Toutes** les instructions suivantes un `case` valide (dont la valeur correspond à la variable) seront exécutées.

```
5      String couleur_feu = "Rouge" ;
6
7      switch( couleur_feu ) {
8
9          case "Vert":
10             System.out.println("Circulez !");
11
12         case "Rouge":
13             System.out.println("Stop !");
14
15     }
16
```

TERMINAL Run: Main

```
Stop !
PS C:\Java>
```

Structure Switch

Du fait, si la première valeur est correcte, **toutes les instructions suivantes**, quelque soit le **case** dans lequel elles se trouvent, seront exécutées.

Ce peut être intentionnel, et c'est d'ailleurs une possibilité du fonctionnement du **switch**, mais il est également possible d'inhiber ce comportement.

```
5      String couleur_feu = "Vert" ;
6
7  ✓   switch( couleur_feu ) {
8
9  ✓       case "Vert":
10          System.out.println("Circulez !");
11
12  ✓       case "Rouge":
13          System.out.println("Stop !");
14
15      }
16
```

TERMINAL Run: Main

```
Circulez !
Stop !
PS C:\Java>
```

Structure Switch

Pour se faire, on utilisera le mot clef **break** pour interrompre l'exécution des instruction et instantanément sortir du **switch**.

```
5      String couleur_feu = "Vert" ;
6
7      switch( couleur_feu ) {
8
9          case "Vert":
10             System.out.println("Circulez !");
11             break;
12
13         case "Rouge":
14             System.out.println("Stop !");
15             break;
16
17     }
```

TERMINAL Run: Main

```
Circulez !
PS C:\Java>
```

Structure Switch

On pourra alors choisir de l'utiliser aux bon endroit pour obtenir un comportement sélectif, comme ici avec le feu orange qui va de paire avec le comportement du feu rouge.



```
5      String couleur_feu = "Orange" ;
6
7      switch( couleur_feu ) {
8
9          case "Vert":
10             System.out.println("Circulez !");
11             break;
12
13          case "Orange":
14          case "Rouge":
15             System.out.println("Stop !");
16             break;
17
18      }
```

TERMINAL

Stop !
PS C:\Java>

Run: Main

Structure Switch

Enfin, il est possible de définir un comportement par défaut dans le cas où aucun **case** ne correspond à la variable.

On utilisera dans ce cas le mot clef **default** suivi des instructions à exécuter dans ce cas.

```
5      String couleur_feu = "Eteint" ;
6
7      switch( couleur_feu ) {
8
9          case "Vert":
10             System.out.println("Circulez !");
11             break;
12
13             case "Orange":
14             case "Rouge":
15                 System.out.println("Stop !");
16                 break;
17
18             default:
19                 System.out.println("Suivre les indications !");
20
21         }
22
```

TERMINAL Run: Main + ▾

Suivre les indications !
PS C:\Java> █

Opérateur ternaire

Derrière ce nom barbare d'« opérateur ternaire » se cache une simplification syntaxique bien pratique.

Le but est ici de pouvoir exprimer une valeur en fonction d'une condition, sans utiliser la structure if assez volumineuse.

On utilisera pour ce faire la syntaxe :

Condition ? **Valeur si vrai** : **Valeur si faux** ;

```
5      int x = 33 ;
6
7      System.out.print("X");
8      System.out.print( x % 3 == 0 ? " est " : " n'est pas " );
9      System.out.print("un multiple de 3");
10
11
```

TERMINAL Run: Main + ▢

X est un multiple de 3
PS C:\Java> █

Opérateur ternaire

```
5      int x = true ? 10 : 20 ;
6
7      String a = (
8          "La variable x est "
9          + (
10             x > 10
11             ? "supérieure"
12             : (
13                 x < 10
14                 ? "inférieure"
15                 : "égale"
16             )
17          )
18          + " à 10"
19      );
20
21      System.out.println(a);
```

Il est possible de l'utiliser dans de nombreux cas comme lors de l'initialisation d'une variable ou pour conditionner la création d'une chaîne de caractères.

Attention cependant à ne pas l'utiliser dans des cas trop complexe. Gardons à l'esprit que l'objectif est de rendre la syntaxe plus lisible.

Ci-contre, la définition de `x` est plus lisible qu'une structure conditionnelle, mais la définition de `a` est devenue lourde à comprendre et nécessiterait plutôt une structure conditionnelle.



Pause questions

Chapitre 6



Les Arrays

Qu'est ce qu'un Array ?

Création d'un array

Lecture d'une case

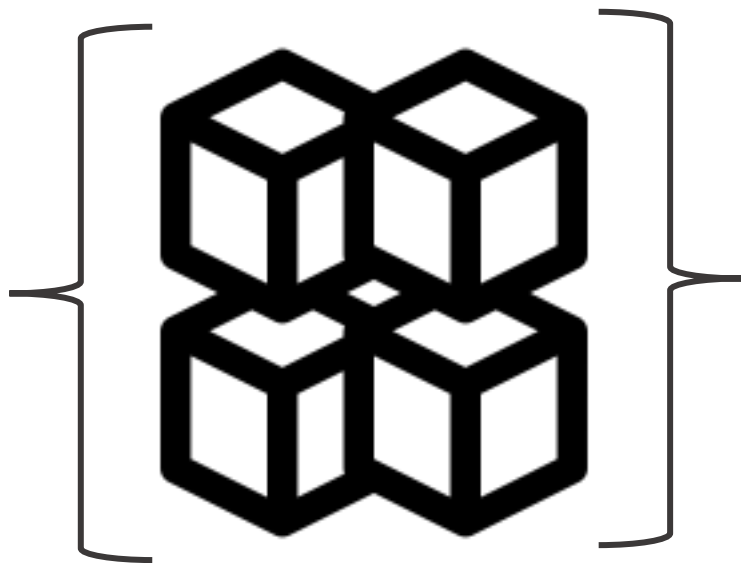
Modification d'une case

Taille d'un array

Tableau multi-dimensionnel

Qu'est-ce qu'un Array ?

Parfois, il peut être utile de stocker dans une variable une **collection d'éléments**. C'est un peu comme si une boîte pouvait contenir plusieurs valeurs, une **boîte à plusieurs emplacements**.



On appelle ces boîtes à plusieurs emplacement des **tableaux**, des **collections** ou encore en Javascript des **Array** (tableaux en anglais).

Création d'un Array

C'est donc un type de données un peu particulier et plus complexe que ceux vus jusqu'à présent. Attention, en Java, un tableau ne peut contenir qu'un seul type de données à la fois.

Pour créer un Array, il suffit de créer une variable dans laquelle le stocker. Elle devra être du type de données que l'on souhaite stocker.

On en définit ensuite le contenu avec les « {} ». Pour le moment nous allons le laisser vide.



```
int[] mes_entiers = {};
```


Les [] indique qu'il s'agit d'une variable de type Array.

Création d'un Array

Pour ajouter des valeurs dans notre tableau, ici des entiers puisqu'il s'agit d'un tableau d'**int**, nous allons les renseigner entre les « {} » séparées chacune par une « , ».

Les différentes valeurs de notre tableau **seront ordonnées**. C'est-à-dire qu'elles auront chacune une position particulière, un numéro de case **en partant de 0, progressant de 1 en 1**.

```
int[] mes_entiers = { 10, 47, -10 };
```



N° de case	0	1	2
Valeurs	10	47	-10

Lecture d'une case

Pour accéder à la valeur d'une case particulière, nous aurons besoins du numéro de cette case, aussi appelé **index**.

En appelant le tableau suivi des « `[]` », nous demandons à récupérer une case précise. Il suffira ensuite d'indiquer l'index de la case souhaitée entre ces crochets pour en obtenir la valeur.

La valeur ainsi récupérée peut être utilisée comme une variable : pour la stocker, effectuer des calculs, l'afficher, etc...

5		<code>int[] mes_entiers = { 10, 47, -10 };</code>
6		<code>System.out.println(mes_entiers[1]);</code>
7		

TERMINAL


47
PS C:\Java> █

Modification d'une case

Pour modifier la valeur d'une des cases de notre tableau, la démarche est identique à la modification d'une variable.

Il nous suffit de récupérer la valeur de la case souhaiter et de lui réaffecter une nouvelle valeur avec l'un des opérateurs d'affectation vus précédemment.

```
int[] mes_entiers = { 10, 2, 5 };  
mes_entiers[1] *= mes_entiers[0];  
mes_entiers[2] = mes_entiers[1] + mes_entiers[2] ;  
mes_entiers[0] = mes_entiers[2] - mes_entiers[0] ;
```



N° de case	0	1	2
Valeurs	?	20	?

Seriez-vous capable de déterminer les valeurs des cases 0 et 2 ?

Taille d'un tableau

Différents tableaux peuvent avoir des tailles variées. Il est donc souvent utile de pouvoir obtenir cette taille.

Tous les tableaux possède une fonction « `.length` » permettant d'en obtenir la taille sous la forme d'un nombre entier.

```
5      int[] mes_entiers = { 10, 2, 5 };
6      int[] autres_entiers = { 0, -6, 80, 76, 3};
7
8      System.out.println( mes_entiers.length );
9      System.out.println( autres_entiers.length );
10
```

TERMINAL Run: Ma

```
3
5
PS C:\Java>
```

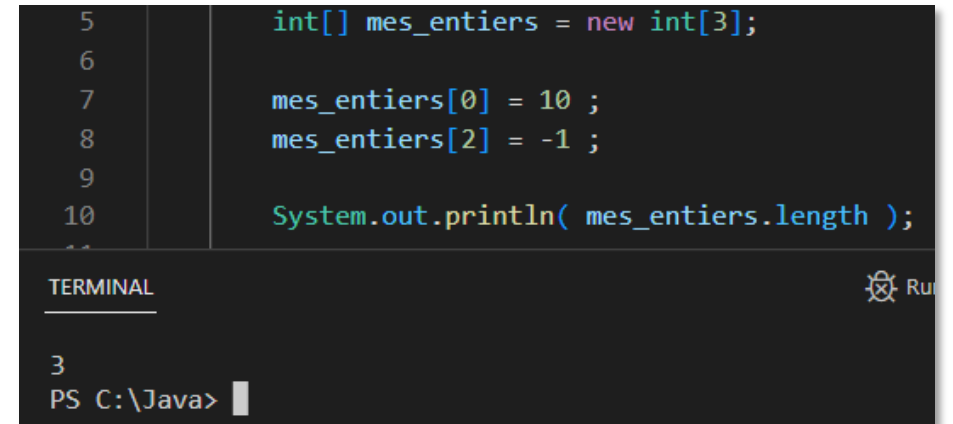
Taille d'un tableau

Lors de la création d'un tableau, si l'on spécifie dès cet instant les valeurs comprises dans ce tableau, alors il sera dimensionné à la juste taille pour pouvoir les contenir.

Dans certains cas, il est nécessaire de définir la taille du tableau d'abord. Pour ce faire, on utilisera la syntaxe suivante .

```
TYPE[] NAME = new TYPE[ TAILLE ] ;
```

Où **TYPE** est le type de tableau, **NAME** le nom de la variable contenant le tableau, et **TAILLE** le nombre de cases désiré.



```
5      int[] mes_entiers = new int[3];  
6  
7      mes_entiers[0] = 10 ;  
8      mes_entiers[2] = -1 ;  
9  
10     System.out.println( mes_entiers.length );  
11  
-----  
TERMINAL  
3  
PS C:\Java>
```

Tableau multi-dimensionnel

Un tableau peut également comporter, dans chacune de ses cases, un autre tableau.

Ainsi, un tableau comportant d'autres tableau sera un tableau à 2 dimensions (ou 2 niveau). Mais on peut aller encore plus loin avec 3, 4 ou une infinité de dimensions.

Pour initialiser un tableau à 2 dimensions, nous devons le typer avec une paire de `[]` par dimension et rassembler les valeurs de chaque tableau dans des `{}` comme ci-contre.

```
int [][] dim2array = {  
    {10, 20, 30},  
    {40, 50},  
    {60, 70, 80, 90}  
};
```

Tableau multi-dimensionnel

Pour accéder aux données d'une case précise, il nous faudra alors parcourir le tableau, dimension après dimension, avec l'opérateur `[]`.

Notez que, bien qu'un tableau, même multi-dimensionnel, ne peut comporter que des valeurs d'un unique type, la taille de chaque tableau peut varier.

```
7      int [][] dim2array = {  
8          {10, 20, 30},  
9          {40, 50},  
10         {60, 70, 80, 90}  
11     };  
12  
13     System.out.println( dim2array[2][3] );  
14     dim2array[2][3] *= dim2array[0][0];  
15     System.out.println( dim2array[2][3] );  
16  
TERMINAL  
  
90  
900
```



Pause questions

Chapitre 7



Structures itératives

Itérations

Structure while

Structure do while

Structure for

Structure for each

Break & continue

Itérations

Dans certaines situation il sera nécessaire de répéter une portion de code.

On utilisera pour cela une structure de répétition, que l'on appelle souvent **structure itérative** ou boucle.

Ces structures sont le **while**, le **for** et le **for in**. Elles ont chacune leur mode de fonctionnement mais peuvent **toutes réaliser les même objectifs**.

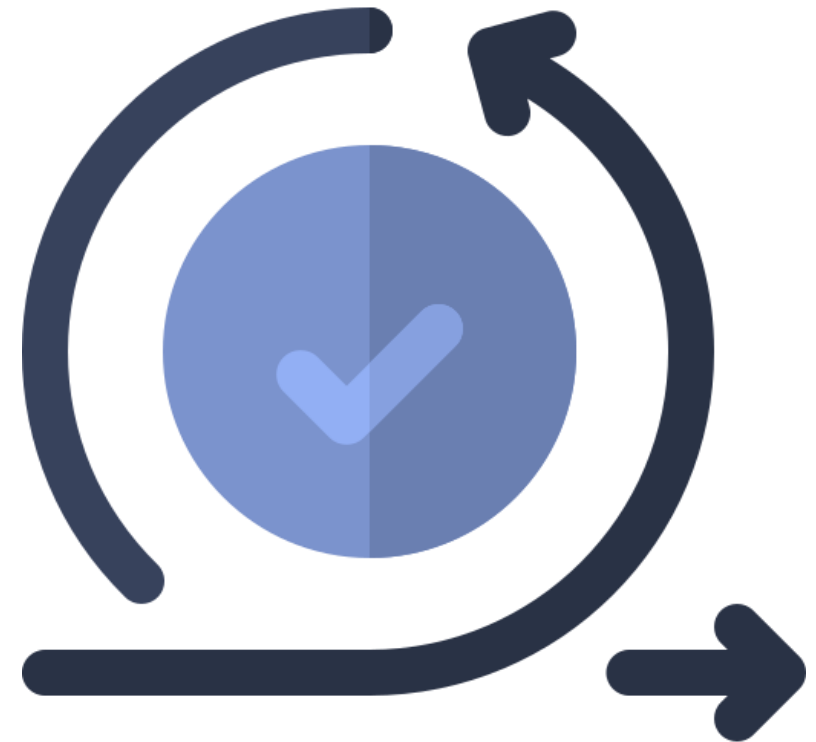


Itérations

Le principe d'une structure itérative est de **répéter un ensemble d'instruction tant qu'une condition** est vraie.

On appelle chacune de ces répétitions **une itération**. Avant chacune d'entre elles, on vérifie que la condition soit vraie.

Lorsque la **condition est fausse**, la **boucle s'arrête**.



Structure While

La structure itérative la plus simple en Java est la structure **while**.

Celle-ci commence par le mot clef **while** suivi par une condition entre **()**.

Le **while** vérifiera tout d'abord la condition, si elle est vraie, on exécutera l'instruction ou segment d'instructions suivant.

```
while( /* condition */ )  
    // Instruction simple à répéter  
  
while( /* condition */ ) {  
    // Ensemble d'instructions  
    // à répéter  
}
```

Structure While

```
int distance = 0 ;

while( distance < 10 ) {
    distance++;
    System.out.println(distance + " km à pied, ça use, ça use...");
}

System.out.println("Arrivé !");
```

```
1 km à pied, ça use, ça use...
2 km à pied, ça use, ça use...
3 km à pied, ça use, ça use...
4 km à pied, ça use, ça use...
5 km à pied, ça use, ça use...
6 km à pied, ça use, ça use...
7 km à pied, ça use, ça use...
8 km à pied, ça use, ça use...
9 km à pied, ça use, ça use...
10 km à pied, ça use, ça use...
Arrivé !
```

Une fois l'itération effectuée, c'est-à-dire le code concerné par le **while**, on vérifie de nouveau la condition et ainsi de suite.

En d'autres termes, tant que la condition n'est pas fausse, on répète l'instruction, ou suite d'instruction.

Dès que la condition est fausse, on passera à la suite, c'est-à-dire à l'instruction suivant la structure **while**.

Structure Do While

La structure **do while** est assez similaire au **while**. Elle répète elle aussi une instruction ou suite d'instruction tant que sa condition est vraie.

La différence intervient dans l'exécution et la syntaxe. On exécutera d'abord les instructions avant d'en vérifier la condition.

Coté syntaxe, on utilisera le mot clef **do** pour annoncer le code à répéter. S'en suivra le **while** et sa condition.

```
int distance = 0 ;

do {
    distance++;
    System.out.println(distance + " km à pied, ça use, ça use...");
} while( distance < 10 ) ;

System.out.println("Arrivé !");
```

Structure Do While

La différence fondamentale entre ces deux structures réside dans la possibilité de ne pas du tout exécuter notre code si la condition est fausse avec le **while**.

```
int distance = 20 ;

do {
    distance++;
    System.out.println(distance + " km à pied, ça use, ça use...");
} while( distance < 10 ) ;

System.out.println("Arrivé !");
```

```
21 km à pied, ça use, ça use...
Arrivé !
```

```
int distance = 20 ;

while( distance < 10 ) {
    distance++;
    System.out.println(distance + " km à pied, ça use, ça use...");
}

System.out.println("Arrivé !");
```

```
Arrivé !
```

Structure For

La structure **for** est elle aussi une structure itérative. Elle peut tout à fait faire le travail d'un **while** mais son objectif est plutôt de parcourir un ensemble de valeurs.

Sa ligne syntaxe se compose du mot clef **for** et de 3 paramètres entre parenthèses, chacun séparé par un « ; ».

A la suite de cette définition, on trouvera, comme pour le **while**, un segment de code à répéter.

```
for( ; ; ) {  
    // Code à répéter  
}
```

Structure For

Le premier paramètre, entre « (» et « ; », est une instruction qui sera exécutée à l'initialisation de la structure.

Le deuxième paramètre, entre les deux « ; », est la condition à laquelle on exécutera le segment de code, à l'instar de la condition d'un **while**.

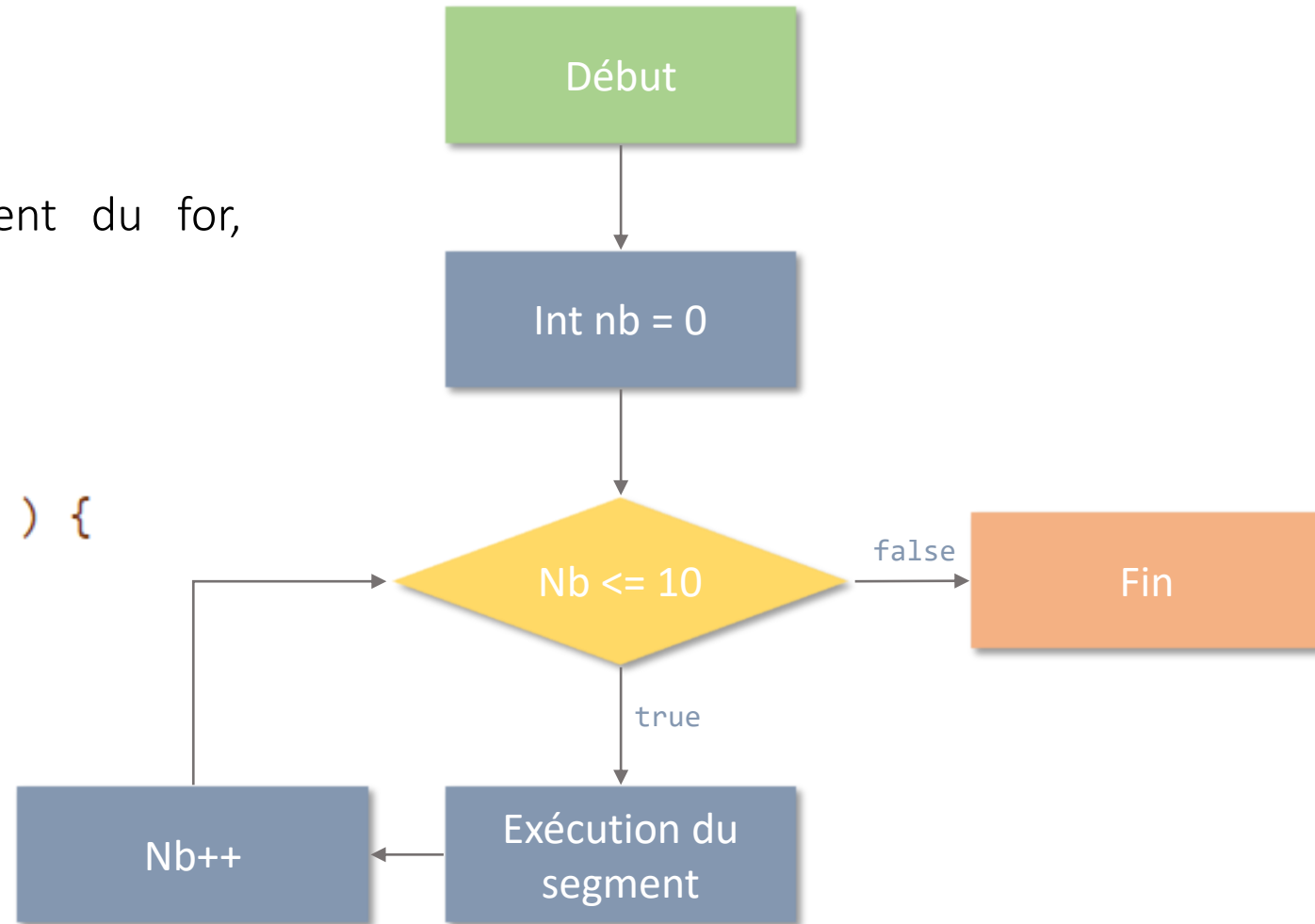
Enfin, en dernier paramètre, entre « ; » et «) », il s'agira d'une instruction à exécuter après chaque itération.

```
// Compter jusqu'à 10 !  
for( int nb = 0 ; nb <= 10 ; nb++ ) {  
    System.out.println( nb );  
}
```

Structure For

Pour bien comprendre le fonctionnement du for, retraçons ses différentes étapes :

```
// Compter jusqu'à 10 !  
for( int nb = 0 ; nb <= 10 ; nb++ ) {  
    System.out.println( nb );  
}
```



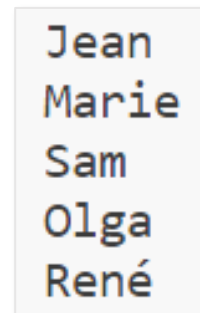
Structure For

La structure for est en général à privilégier lorsque l'on a un point de départ et un point d'arrivée connus pour notre répétition.

On l'utilisera d'ailleurs le plus souvent pour le parcours des différents éléments d'un ensemble, comme un **array** par exemple.

Pour parcourir les valeurs d'un ensemble, on créera alors un index parcourant les différents numéros de cases.

```
String[] names = { "Jean", "Marie", "Sam", "Olga", "René" } ;  
  
for( int index = 0 ; index < names.length ; index++ ) {  
    System.out.println( names[ index ] );  
}
```



Jean
Marie
Sam
Olga
René

Structure Foreach

La structure for permet également l'usage d'une syntaxe simplifiée pour le parcours de valeurs.

Il est possible de se passer d'un index et de récupérer directement chaque valeur lors de chaque itération.

On définira, dans les « `()` », une variable adaptée pour la capture de chaque valeurs puis l'ensemble à parcourir séparée par un « `:` ».

```
String[] names = {  
    "Jean", "Marie", "Sam",  
    "Olga", "René"  
} ;  
  
for( String name : names ) {  
    System.out.println( name );  
}
```

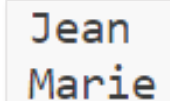
Break & Continue

Lors de l'exécution d'une structure itérative, **while** ou **for**, il sera possible d'utiliser les instructions **break** et **continue**.

L'instruction **break**, lorsqu'elle sera lue, interrompra la boucle dans sa totalité.

Cela signifie donc que l'on passera directement aux instructions suivant la boucle.

```
String[] names = {  
    "Jean", "Marie", "Sam",  
    "Olga", "René"  
};  
  
for( String name : names ) {  
    if( name.equals( "Sam" ) ) {  
        break ;  
    }  
    System.out.println( name );  
}
```



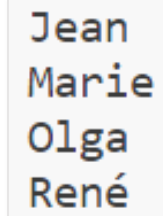
Jean
Marie

Break & Continue

L'instruction **continue** quant à elle permettra de mettre fin à l'itération en cours.

L'instruction **continue** ne met donc pas systématiquement fin à la boucle, mais passe à l'itération suivante.

```
String[] names = {  
    "Jean", "Marie", "Sam",  
    "Olga", "René"  
} ;  
  
for( String name : names ) {  
    if( name.equals( "Sam" ) ) {  
        continue ;  
    }  
    System.out.println( name );  
}
```



```
Jean  
Marie  
Olga  
René
```



Pause questions

Chapitre 8



Les fonctions *(ou méthodes)*

Les fonctions en Java

Syntaxe d'une fonction

Exécuter une fonction

Les paramètres

La valeur de retour

La fonction main


Paramètres infinis

Surcharge

Les fonctions *(ou méthodes)*


En programmation, une fonction, ou parfois appelée procédure, regroupe un ensemble d'instruction sous un même nom.


C'est un peu comme une recette de cuisine : elle porte un nom comme « *Tarte à la framboise* » et regroupe un ensemble d'instructions pour parvenir à un résultat.


 **Préparation**

Temps total: 50 min


Préparation: 20 min	Repos: -	Cuisson: 30 min
------------------------	-------------	--------------------


ÉTAPE 1 
Déroulez la pâte brisée avec sa feuille de cuisson dans un moule à tarte. Faites cuire à blanc 20 minutes à 180°C (thermostat 6).

ÉTAPE 2 
Pendant ce temps, travaillez les oeufs et le sucre. Incorporez la farine et le lait chaud. Faites épaissir le mélange dans une casserole à feux doux en tournant avec une cuillère en bois.

ÉTAPE 3 
Retirez du feu dès que la crème est prise. Ajoutez le sachet de sucre vanillé.

ÉTAPE 4
Sortez le fond de tarte du four. Laissez refroidir. Versez-y la crème.

ÉTAPE 5 
Disposez les framboises en cercle sur la crème, pour faire une belle présentation.

ÉTAPE 6 
Nappez de gelée de framboises liquéfiée avec 2 cuillerées d'eau.

Les fonctions *(ou méthodes)*

Le langage Java est un langage dit « orienté objet ». Cela impacte notamment la façon de concevoir des fonctions.

Nous en verrons les détails plus tard, mais en Java, nos fonctions prendront également le nom de méthodes.



Les fonctions *(ou méthodes)*

Nous connaissons d'ailleurs une méthode dans laquelle nous avons beaucoup travaillé : la méthode Main.

Il existe aussi d'autres fonctions comme **equals** pour les chaînes ou **println**.

```
public static void main( String[] args ) {  
  
    if( password.equals("p4ssw0rd") ) {  
        System.out.println("Connected !");  
    }  
  
}
```

Syntaxe d'une fonction

Pour définir une fonction, nous devons nous placer à l'extérieur de la fonction **Main**, c'est à dire dans la partie que l'on appelle la classe.

Dans l'exemple ci-contre, on pourra écrire nos fonctions avant ou après la fonction **main**.

Notez qu'on appelle également la création d'une fonction une déclaration de fonction.

```
public class Main {  
    /*  
     Déclaration des fonctions  
    */  
  
    public static void main( String[] args ) {  
  
    }  
}
```

Syntaxe d'une fonction

Pour déclarer une fonction, nous devons tout d'abord en écrire le prototype.

Il s'agit d'une ligne permettant d'en définir plusieurs éléments comme le nom.

Ci-contre, nous définissons une nouvelle fonction « **ma_fonction** » qui sera pour le moment vide.

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

Syntaxe d'une fonction

Nous pouvons ensuite lui fournir les instructions à exécuter en cas d'appel.

Ces différentes instructions seront placées entre les « {} ».

```
public static void ma_fonction() {  
    System.out.println("Bonjour utilisateur !");  
    System.out.println("On programme en Java ?");  
}
```

Exécuter une fonction

Pour exécuter notre fonction, nous pourrions l'appeler par son nom. Il faudra ensuite lui ajouter des « () », vides pour le moment.

Ainsi, à l'endroit on nous avons effectué cet appel, les instructions de notre fonction vont s'exécuter.

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

```
Bonjour utilisateur !  
On programme en Java ?
```

Exécuter une fonction

Les intérêts sont nombreux. Outre la simplification du code en nommant votre ensemble d'instructions, cela permet de réutiliser cette procédure n'importe où.

En terme de maintenance, c'est également un gain de temps. Si une correction est apportée, elle le sera pour tous les appels.

```
public static void main( String[] args ) {  
    for( int i = 0 ; i < 5 ; i++ ) {  
        ma_fonction();  
    }  
}
```

```
Bonjour utilisateur !  
On programme en Java ?  
Bonjour utilisateur !  
On programme en Java ?  
Bonjour utilisateur !  
On programme en Java ?  
Bonjour utilisateur !  
On programme en Java ?  
Bonjour utilisateur !  
On programme en Java ?
```

Les paramètres

Dans certains cas, notre fonction aura besoins de paramètre, c'est-à-dire de valeurs extérieures, pour fonctionner.

Cela permet de rendre dynamique notre fonction, elle s'adapte aux différentes valeur qu'on pourra lui fournir.

Les paramètre devront être définis dès la création de la fonction entre les « () ».

```
public static void ma_fonction( /* Paramètres */ ) {  
  
}
```

Les paramètres

Ces paramètres devront être typés et nommé.
Il s'agira en réalité d'une déclaration de variable locale à notre fonction.

Dans l'exemple ci-contre, nous créons une fonction « **dire_bonjour** » dont l'objectif est de dire bonjour à « **nom_utilisateur** » « **n** » fois.

```
public static void dire_bonjour(  
    String nom_utilisateur,  
    int n  
) {  
    for( int i = 0 ; i < n ; i++ ) {  
        System.out.println("Bonjour " + nom_utilisateur);  
    }  
}
```


Les paramètres

Lors de l'utilisation de notre fonction, nous devons alors fournir des valeurs cohérentes avec les types proposés plus tôt.

Chaque usage de notre fonction peut se faire avec des valeurs différentes.

```
public static void main( String[] args ) {  
    dire_bonjour( "Albert", 2 );  
    dire_bonjour( "Alice", 1 );  
    dire_bonjour( "Marie", 3 );  
}
```

```
Bonjour Albert  
Bonjour Albert  
Bonjour Alice  
Bonjour Marie  
Bonjour Marie  
Bonjour Marie
```

La valeur de retour

Dans certains cas, une fonction ne se contentera pas d'effectuer certaines actions.

Elle pourra également retourner une valeur à son appelant. Pour ce faire, on utilisera le mot clef **return** suivi de la valeur à retourner.

Attention, **return** met fin instantanément à l'exécution de la fonction, c'est donc la dernière action à effectuer.

```
public static double surface_cercle( double rayon ) {  
    return 3.14 * rayon * rayon ;  
}
```

```
public static void main( String[] args ) {  
    System.out.println(  
        "La surface d'un cercle de rayon 4 est de "  
        + surface_cercle( 4.0 )  
    );  
}
```

La valeur de retour

Il est ensuite possible de réutiliser cette valeur pour n'importe quel usage, comme s'agissait d'une valeur brute ou d'une variable.

Par exemple, ci-contre, on réutilisera notre retour de la fonction `surface_cercle` comme paramètre pour le calcul d'une autre fonction.

Enfin, on affichera le résultat de cette nouvelle fonction `volume_cylindre` par le biais de la fonction `System.out.println`.

```
public static double surface_cercle( double rayon ) {  
    return 3.14 * rayon * rayon ;  
}  
  
public static double volume_cylindre( double surface_face, double hauteur ) {  
    return surface_face * hauteur ;  
}  
  
public static void main( String[] args ) {  
    System.out.println(  
        "Le volume d'un cylindre de 4 de rayon et de 6 de hauteur est de "  
        + volume_cylindre( surface_cercle( 4.0 ), 6.0 )  
    );  
}
```

```
Le volume d'un cylindre de 4 de rayon et de 6 de hauteur est de 301.44
```

La fonction main

Et la fonction main dans tout ça ?

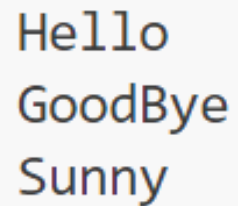
Il s'agit en fait d'une fonction exécutée lors du lancement du programme. C'est donc notre cheminement principal.

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

Elle prend également un paramètre un peu particulier `String [] args`. Il s'agit d'un tableau de chaînes de caractères.

La fonction main

```
public static void main( String[] args ) {  
    for( String arg : args ) {  
        System.out.println( arg );  
    }  
}
```



Hello
GoodBye
Sunny

Ces chaînes de caractères sont en fait les différents paramètres que recevra le programme lors de son exécution.

Il est alors possible de les lire et de s'en servir comme bon vous semble

Paramètres infinis

Tout comme les arguments de la fonction main, il est possible de définir un nombre infini de paramètres d'un certains type à nos fonctions.

Il faudra pour cela intercaler des points de suspension « ... » entre le type et le nom du paramètre.

```
public static int sum( int ... nbs ) {  
    int total = 0 ;  
    for( int nb : nbs ) {  
        total += nb ;  
    }  
    return total ;  
}  
  
public static void main( String[] args ) {  
    System.out.println( sum(10, 20, 30 ) );  
}
```



Pause questions