Chapitre 2 : Les bases de Java

▼ Les variables

Qu'est-ce qu'une variable ?

Dans les langages de programmation, une **variable** associe un **nom** (ou identifiant) avec une **valeur**. Dans la machine, cela veut dire qu'un **espace mémoire est réservé pour stocker** une valeur.

Dans la plupart des langages, il est autorisé de **changer** la valeur d'une variable au cours du temps. Il est également possible d'être plus **restrictif** et de **ne pas autoriser le changement** de valeurs.

Les variables sont techniquement stockées dans la **mémoire vive** (**RAM : Random Access Memory**). Les variables d'un programme dans la mémoire vive ne sont **utilisables que pendant l'exécution d'un programme**. C'est un type de mémoire pensé pour avoir **l'accès** aux valeurs du programme le plus **rapidement** possible.

Les données des **variables** ne sont donc pas **persistantes**, contrairement à la **mémoire morte (ROM : Read Only Memory)**. Celle-ci est utilisée pour stocker de la donnée sur un ordinateur de manière persistante, mais son **accès** est plus **lent** que la **mémoire vive**.

Déclarer une variable

La syntaxe pour déclarer une variable est la suivante :

```
[type] [nom_variable] = [valeur];
```

- `[type]` : le type de données de la variable (par exemple : `byte`, `int`, `String`, etc.).
- `[nom_variable]` : le nom de la variable.
- `[valeur]` : la valeur initiale de la variable (optionnelle pour certaines variables comme les `int`, `byte`, `long`, etc.).

Donc, la déclaration de la variable `age` en tant que `byte` avec une valeur de `10` serait :

```
byte age = 25;
```

Cela crée une variable nommée `age` de type `byte` et lui assigne la valeur `10`.

Java est un langage **typé**, il est donc obligatoire de commencer par définir son type. Ici, on déclare **un entier de 1 octet**.

Ensuite, le nom de la variable est défini. Les règles suivantes doivent être respectées:

- **On peut utiliser des lettres pour les variables.**
- **On ne peut pas utiliser un chiffre en première position (`1age` interdit).**
- **On peut utiliser un chiffre partout ailleurs (`age1` autorisé).**

- **On ne peut rien utiliser d'autre.**

Convention de nommage

Dans la communauté `Java`, les développeurs respectent une **convention de nommage** pour les variables : le `camelCase`.

Votre code fonctionne techniquement sans respecter cette convention, mais il est très **fortement conseillé** de suivre cette **convention**. Dès lors que **vous travailler sur du code avec d'autres développeurs**, le **respect** des conventions permet de faciliter la lecture du code.

Exemple de nom de variable en **camelCase** : `nomEnCamelCase`.

Le **premier mot de la variable** est totalement en **minuscule** et les **mots suivants** voient leur **première lettre** s'afficher en **majuscule**.

Inférence

Si le **type** est **obligatoire**, il est également possible de laisser le **compilateur inférer** le type de la variable comme dans l'exemple suivant :

```
var age = 25;
```

Ici, le **compilateur** a les informations nécessaires pour **deviner** le type. Il va **automatiquement** définir le type de `age` à `int`, **un entier de 4 octets**.

Exemple:

```
public class Main {
    public static void main(String[] args) {
        var age = 25;
        Object o = age;
        System.out.println(o.getClass());

    }
}
//Console
┌─² pwsh ᵧ D:\COURS\Java\Code
class java.lang.Integer
```

La plupart du temps, nous pouvons utiliser le keyword var pour déclarer des variables, cela permettra d'écrire du code plus concis (ce sera plus vrai quand nous verrons des concepts plus avancés dans les chapitres suivants).

Afficher le contenu d'une variable

L'exemple suivant montre comment afficher le contenu d'une variable dans la console :

```
int resultat = 1500;
System.out.println(String.format("Contenu de resultat : %s", resultat));
//Console
Contenu de resultat : 1500
```

La mémoire : Heap et Stack

> Comprendre cette partie n'est pas nécessaire pour coder en Java. La lire vous permettra de comprendre comment les variables Java sont techniquement stockées.

Les variables à type natif ou primitif

Dans ce chapitre, les **types natifs seront utilisés**.

À chaque déclaration de variable native, un emplacement mémoire est créé dans la `stack` **avec une adresse**, et la valeur de la variable est **stockée** dans cet **emplacement**.

Les **avantages** de la `stack` :

Les **performances d'accès** de la `stack` sont **meilleurs** que celles de la `heap`. Il est donc plus avantageux de l'utiliser plutôt que la `heap` quand c'est possible.

Les variables objet

Nous utiliserons ces variables dans le chapitre : `Programmation orientée objet`.

Les variables **objet** prennent également un emplacement mémoire dans la `stack`. Cependant, la valeur est **stockée** dans la `heap` et celle-ci est **référencée** par **l'emplacement mémoire(adresse)** de la `stack` de la variable. La `stack` **ne stocke donc pas la valeur** de la variable mais une **référence**.

L'utilisation de la `heap` est très utile pour avoir des variables dont la **taille varie** et dont la **durée de vie est assez longue** (cela arrive quand un même objet est utilisé par plusieurs bouts de code).

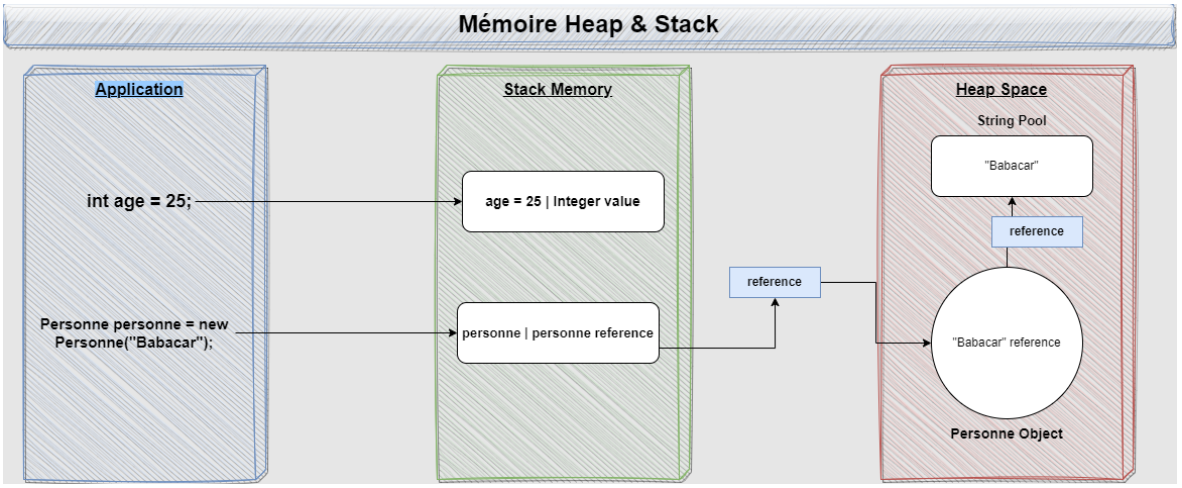

Illustration gestion mémoire Heap & Stack

Le garbage collector

`Java` est un langage de **haut niveau**. Si vous vous rappelez, la gestion de la mémoire est automatique et le développeur ne s'occupe pas de libérer la mémoire utilisée par les variables.

C'est la raison d'être du **garbage collector**. Il va s'occuper de détecter pour nous toutes les variables qui ne sont plus utilisées par le programme et va donc rendre disponibles de nouveau les **emplacements mémoires** utilisés par celles-ci.

▼ Les types

Dans les leçons précédentes, nous avons vu que Java est un langage fortement typé. Lors de la déclaration des variables, il est donc obligatoire de définir le type de celles-ci.

Au niveau syntaxe, Java est un langage de programmation qui ressemble beaucoup au langage

`c++`. Toutefois quelques simplifications ont été apportées à java pour des raisons de sécurité et d'optimisation.

Dans cette leçon, nous allons voir quels sont les types natifs de Java.

Java dispose des primitives suivantes :

Primitive	***Étendue***	***Taille***
char	0 à 65 535	16 bits
byte	-128 à +127	8 bits
short	-32 768 à +32 767	16 bits
int	-2 147 483 648 à + 2 147 483 647	32 bits
long		64 bits
float	de ± 1.4E-45 à ± 3.40282347E38	32 bits
double		64 bits
boolean	true ou false	1 bit

Quel est l'intérêt de forcer le typage des variables ?

Cela peut paraître très contraignant aux premiers abords, mais forcer le typage des variables comprend quelques avantages :

- On est **sûr** que la **valeur** d'une **variable correspond** au **type attendu**. Sinon, on obtient une erreur à la compilation.
- On **évite** donc des **erreurs potentielles** à l'exécution, on **réduit** ainsi le risque de **bugs** sur le **long terme**.
- Quand on **valide** les paramètres d'entrées d'un programme ou d'une fonction, moins de tests de vérifications liés au type des variables sont nécessaires car la **compilation assure déjà beaucoup de vérifications pour nous.**

Par exemple, il est impossible d'assigner une **chaîne de caractères** à une variable de type **entier**

```
int notAString = "will fail";
//Erreur de compilation car impossible d'assigner une chaîne de caractère
//à une variable de type entier
```

Les entiers

Permet la définition de variables correspondant à **l'ensemble mathématique des entiers**. Cela correspond par exemple aux valeurs suivantes : **-22**, **0**, **42**.

Il existe différents types d'entiers en fonction de la taille minimum ou maximum dont on a besoin lors de l'utilisation du programme :

- `byte` : **8 bits ⇒ de -128 à 127**
- `short` : **16 bits ⇒ de -32768 à 32767**
- `int` : **32 bits ⇒ de -2.147.483.648 à 2.147.483.647**
- `long` : **64 bits ⇒ de -9.223.372.036.854.775.808 à 9.223.372.036.854.775.807** (Neuf quintillions deux cent vingt-trois quadrillions trois cent soixante-douze trillions trente-six billions huit cent cinquante-quatre milliards sept cent soixante-quinze millions huit cent sept mille huit cent sept)

Exemple de déclaration **d'entier** :

```
long entierLong = 9223372036775807;
```

Les nombres à virgule flottante

Permet la définition de variables correspond à l'ensemble mathématiques des réels. Cela correspond par exemple aux valeurs suivantes : **-22.3**, **0.11**, **42.42.**

> **NB: En informatique la virgule est exprimé par point(.)**

Il existe différents types de flottants en fonction de la taille minimum ou maximum dont on a besoin lors de l'utilisation du programme :

- `float` : **32 bits ⇒ de 1.401e-045 à 3.40282e+038**
- `double` : **64 bits ⇒ de 2.22507e-308 à 1.79769e+308**

Exemple de déclaration de **flottants** :

```
float flottant = 5233.434
```

Les caractères

Permet la représentation de n'importe quel caractère. Un caractère est représenté par un code Unicode (liste des caractères). Dans les faits, on utilise la plupart du temps les caractères directement présents sur nos claviers comme dans l'exemple suivant :

```
char lettreA = 'a';
```

> **NB: A la déclaration les caractères sont entourés par des simples quottes : `'`**

Il n'existe qu'un **seul** type de caractères : `char`.

Cas particulier : les chaînes de caractères

Permet la représentation d'une `concaténation de caractères`.

Une chaîne de caractères n'est pas un **type natif de Java**, mais un `objet`. Sans rentrer dans les détails, son utilisation ressemble quand même à un type natif car sa déclaration se fait de la même manière comme le montre l'exemple suivant :

```
String chaine = "Une chaîne de caractères";
var avecInference = "Quand même une chaîne de caractères";
```

Le type d'une chaîne de caractères est le suivant : `String`.

Les booléens

Permet la représentation d'une valeur **logique** qui correspond à une seule des deux valeurs suivantes : `true` ou `false`.

Il n'existe qu'un seul type de booléen : `boolean`.

Ce type est très utile pour faire des **opérations logiques afin de vérifier des conditions** dans des programmes.

Exemple de déclaration d'un booléen :

```
boolean bool = true;
boolean boolFalse = false;
```

Les enveloppeurs (Wearpers)

Les primitives sont enveloppées dans des objets appelés **enveloppeurs** (`Wearpers`).

Les enveloppeurs sont des **classes.**

Classe	Primitive
`Character`	char
`Byte`	byte
`Short`	short
`Integer`	int
`Long`	long
`Float`	float
`Double`	double
`Boolean`	boolean
`Void`	-
`BigInteger`	-
`BigDecimal`	-
`String`	-

Exemple:

```
double v1=5.5; // v1 est une primitive
Double v2=new Double(5.6); // v2 est un objet
long a=5; // a est une primitive
Long b=new Long(5); // b est un objet
Long c= 5L; // c est un objet

System.out.println ("a="+a);
System.out.println("b="+b.longValue());
System.out.println("c="+c.byteValue());
System.out.println("V1="+v1);
System.out.println("V2="+v2.intValue());

Résultat:
a=5
b=5
c=5
V1=5.5
V2=5
```

Casting des primitives

Le **casting** (mot anglais qui signifie moulage), également appelé **cast** ou parfois **transtypage**, consiste à effectuer une conversion d'un type vers un autre type.

Le casting peut être effectué dans deux conditions différentes.

Le `surcasting` et le `souscasting` sont des opérations courantes en Java pour convertir des variables d'un type à un autre. Voici une explication simple de chacun :

1. **Surcasting (Sur-typage)** :
   Le surcasting se produit lorsque vous **convertissez** une variable d'un type de données de **taille inférieure** à un type de données de **taille supérieure**. Cela se fait **implicitement** par le **compilateur** car il est sûr de ne pas perdre de données lors de la conversion ou bien **explicitement** par le programmeur.

   Exemple :

   ```
   int a=6; // le type int est codé sur 32 bits
   long b; // le type long est codé sur 64 bits
   ```

```
Casting implicite: b=a;
Casting explicite b=(long)a;
```

2. **Souscasting (Sous-typage)** :
   Le
   souscasting se produit lorsque vous **convertissez** une variable d'un type de données de **taille supérieure** à un type de données de **taille inférieure**. Cela doit être fait **explicitement** par le programmeur et peut entraîner **une perte de données** si la valeur ne rentre pas dans la **plage** du type de données cible.

   Exemple :

   ```
   double a= 10.5;
   int b = (int) a; // Souscasting explicite: double to int
   ```

   Dans cet exemple, la variable **a** de type **double** est convertie en une variable de type **int**.
   L'opérateur de casting (int) est utilisé pour effectuer
   explicitement le souscasting.
   Notez que la
   partie décimale de la variable **a** est **tronquée** lors de la conversion en **int**.

   > Il est important de noter que le souscasting peut entraîner une perte de précision ou une perte de données si la valeur de la variable dépasse la plage du type de données cible.

Les commentaires

Lors de cette leçon, des `commentaires` ont été utilisés. Les `commentaires` ne sont pas **interprétés pendant la compilation**, il est donc possible d'écrire du texte qui sert d'indication uniquement, mais qui n'aura pas **d'influence** sur le **code exécuté**.

Syntaxe des commentaires en Java :

```
//Commentaire sur une seule ligne

/*
Commentaire
sur
plusieurs
lignes
*/

int entier = 2; //Commentaire sur la même ligne que du code
```

Attention à ne pas utiliser trop de commentaires dans votre code. On dit qu'un code bien écrit se décrit lui-même et n'a pas besoin de commentaires. Veillez à n'ajouter des commentaires que quand ceux-ci ont vraiment de la valeur ajoutée.

▼ Les opérateurs

Les opérateurs sont des symboles qui permettent la manipulation des variables. Cela veut dire qu'il est possible de réaliser tout un panel d'opérations directement sur les variables d'un programme.

> Note : Les opérations possibles dépendent du type des variables concernées.

Il existe trois types d'opérateurs différents :

- Les opérateurs **unaires**
  - Applicable sur un seul élément (par exemple `-45`)
- Les opérateurs **binaires**
  - Applicable sur deux éléments (par exemple `5 + 2`)
- Les opérateurs **ternaire ? :**
  - Applicable sur trois éléments. Nous verrons son utilisation dans le chapitre sur les conditions.

Nous n'allons pas couvrir la totalité des opérateurs dans cette leçon. Le but est de se concentrer les cas les plus usuels, vous aurez l'occasion de rencontrer les opérateurs restants par la suite.

Les opérateurs de calcul

Concerne les opérateurs : `+`, `-`, `*`, `/` et `%`.

Utilisable sur les types : `byte`, `short`, `int`, `long`, `float`, `long`,, `double`, `char`.

Le résultat des opérations de calcul se résolvent de la même manière que les calculs mathématiques :

```
System.out.println(12 + 2); // = 14
System.out.println(12.2 - 2.2); // = 10.0
System.out.println(12 * 3); // = 36
System.out.println(12 / 2); // = 6
System.out.println(22 % 3); // = 1 (Le reste de la division)
System.out.println('d' + 'ù'); // = 349
System.out.println((char) ('d' + 'ù')); // = ŝ en tant que caractère via un cast
System.out.println((char) 68);// D
```

> L'opérateur `%` est le modulo. Le résultat est le reste de la division

> Il est possible de faire des calculs sur les caractères car les valeurs numériques des caractères sont alors utilisées pour les calculs.
> Voir tableau code ASCII

Les opérateurs d'assignation

Concerne les opérateurs : `=`, `+=`, `-=`, `*=`, `/=` et `%=`.

Utilisable sur les types : `byte`, `short`, `int`, `long`, `float`, `long`, `float`, `double`, `char`.

```
var number = 25.3;
number += 3; // Equivalent à number = number + 3
number -= 4.2; // Equivalent à number = number - 4.2
number *= -2; // Equivalent à number = number * -2
number /= 3; // Equivalent à number = number / 3
number %= 2; // Equivalent à number = number % 2
```

En bref, ces **opérateurs** fonctionnement d'une manière similaire aux **opérateurs** de calcul mais avec une réassignation de la variable à gauche de l'expression.

Cas particulier : Incrémenter et décrémenter

Lorsque l'on souhaite incrémenter et décrémenter une variable, il est possible d'utiliser les opérateurs : `++` et `--`.

On distingue deux aspects pour chacun des ces concepts:

- **Pré-incrémentation ou Pré-décrémentation (++i ou - -i):**

- La valeur de `i` est d'abord incrémentée ou décrémentée de 1.
- Ensuite, la nouvelle valeur de `i` est affichée.

- **Post-incrémentation ou Post-décrémentation:(i++ ou i- -):**
  - La valeur actuelle de `i` est d'abord affichée.
  - Ensuite, `i` est incrémenté ou décrémenté de 1.

```
//Pré-incrément
int i = 5;
System.out.println("++i = " + ++i);// Affiche: i = 5
int j = i; // j prend la valeur de i après son incrémentation
System.out.println("j = " + j); // Affiche: j = 6

//Post-décrément
int a = 5;
System.out.println("a++ = " + a++);// Affiche: b = 5
int b = a; // b prend la valeur de a après son incrémentation
System.out.println("b = " + b); // Affiche: b = 4
```

Les opérateurs de comparaison

Concerne les opérateurs : `==`, `<`, `<=`, `>`, `>=` et `!=`.

Utilisable sur les types : `byte`, `short`, `int`, `long`, `float`, `long`, `float`, `double`, `char`.

Ces opérateurs permettent de comparer deux variables de différentes manières, et le résultat de la comparaison sera un booléen :

```
var plusPetit = 4;
var plusGrand = 8;
var testEgalité = plusPetit == plusGrand; // false car différents
var testPlusPetit = plusGrand < plusPetit; // false car pas strictement plus petit
var testPlusPetitOuEgal = 8 <= plusGrand; // true car plus petit ou égal
var testPlusGrand = plusGrand > plusPetit; // true car strictement plus grand
var testPlusGrandOuEgal = plusGrand >= 8; // true car plus grand ou égal
var testDifférence = plusPetit != plusGrand; // true car différents
```

Les opérateurs logiques

Concerne les opérateurs : `||`, `^`, `&&` et `!`.

Utilisable uniquement avec le type `boolean`.

Ces opérateurs permettent de faire des opérations logiques sur des booléens :

```
var ouLogiqueInclusif = false || false; // = false car aucun true
ouLogiqueInclusif = true || false; // = true car au moins un true
ouLogiqueInclusif = false || true; // = true car au moins un true
ouLogiqueInclusif = true || true; // = true car au moins un true

var ouLogiqueExclusif = false ^ false; // = false car aucun true
ouLogiqueExclusif = true ^ false; // = true car un seul true
ouLogiqueExclusif = false ^ true; // = true car un seul true
ouLogiqueExclusif = true ^ true; // = false car plus qu'un seul true

var etLogique = false && false; // = false car pas deux true
etLogique = true && false; // = false car pas deux true
```

```
etLogique = false && true; // = false car pas deux true
etLogique = true && true; // = true car deux true

var condition = true;
var vrai = condition == true; // = true car la condition est bien égale à true
var faux = !condition == true; // = false car la condition est inversée par !
```

Les opérateurs spécifiques aux chaînes de caractères

Il est possible d'utiliser l'opérateur `+` sur les chaînes de caractères :

```
var assalamuAlaykum = "Assalamu" + " alaykum!"; // = Assalamu alaykum!
```

Cela a pour effet de concaténer les chaînes ensembles.

Egalité des chaînes

Utiliser l'opérateur `==` pour tester l'égalité de deux chaînes ne fonctionne pas dans le cas suivant :

```
var salam = "Salam";
var salam2 = new String("Salam");// Création d'un nouvel objet avec new

System.out.println(salam == salam2);// false car considérés comme
//deux objets différents

System.out.println(salam.equals(salam2)); // true car equals compare
//deux contenus égaux
```

Cela s'explique car les **chaînes de caractères** ne sont pas des **types natifs** mais des **objets**, il n'est donc pas possible d'appliquer les opérateurs de comparaison directement (ils ne partagent pas forcément la même référence dans ce cas).

Priorités des opérateurs

Le tableau ci-dessous décrit la liste des opérateurs Java par ordre de priorités (le premier étant le plus prioritaire et le dernier le moins prioritaire) :

les opérateurs postfix	expr++ expr--
les opérateurs unaires	++expr --expr +expr -expr ~ !
les opérateurs de multiplication, division et modulo	* / %
les opérateurs d'addition et soustraction	+ -
les opérateurs de décalage	<< >> >>>
les opérateurs de comparaison	< > <= >= instanceof
les opérateurs d'égalité	== !=
l'opérateur OU exclusif	^
l'opérateur ET	&
l'opérateur OU	|
l'opérateur ET logique	&&
l'opérateur OU logique	||
l'opérateur ternaire	? :
les opérateurs d'assignement	= += -= *= /= %= ^= |= <<= >>= >>>=
l'opérateur arrow	->

Le but n'est pas ici de connaitre par cœur les règles de **priorités** mais plutôt de s'y **référer quand c'est nécessaire**.

Les parenthèses

Si l'ordre d'exécution des **opérateurs** ne vous convient pas, il est possible d'utiliser les parenthèses pour changer celui-ci :

```
var calcul = 12 + 3 * 2; // = 18 car * est prioritaire
var calcul2 = (12 + 3) * 2; // = 30 car les parenthèses sont prioritaires
```

Division par 0

Les entiers

Lors de la division des entiers par 0 :

```
var result = 10 / 0;
```

Ce calcul résulte en une exception, `ArithmeticException: / by zero.`

Dans le cas des entier, la division par 0 n'est donc **pas autorisée et elle provoque une erreur d'exécution**. Il faut donc faire très attention à ne pas avoir de division par 0 lors de la manipulation des entiers car cela peut devenir un **bug**.

Les flottants

Lors de la division des flottants par 0 :

```
var result = 10.0 / 0.0;
```

Le résultat est alors `Double.POSITIVE_INFINITY` (ou `Float.POSITIVE_INIFINITY`).

Selon les calculs, il existe **trois** valeurs différentes pour les flottants (applicable à Float et **Double**) :

- Indéfini : `NaN`
- Indéfini positif : `POSITIVE_INFINITY`
- Indéfini négatif : `NEGATIVE_INFINITY`

Ci-dessous se trouve le tableau récapitulatif des différentes combinaisons possibles avec ces trois valeurs :

X	Y	X / Y	X % Y
valeur finie	0	+ ∞	NaN
valeur finie	+/- ∞	0	x
0	0	NaN	NaN
+/- ∞	valeur finie	+/- ∞	NaN
+/- ∞	+/- ∞	NaN	NaN

Ces comportements avec les flottants sont l'application de la norme IEEE754. Cela permet de représenter les valeurs non finies.