DE PYTHON À JAVA

IFT1025 - PROGRAMMATION 2

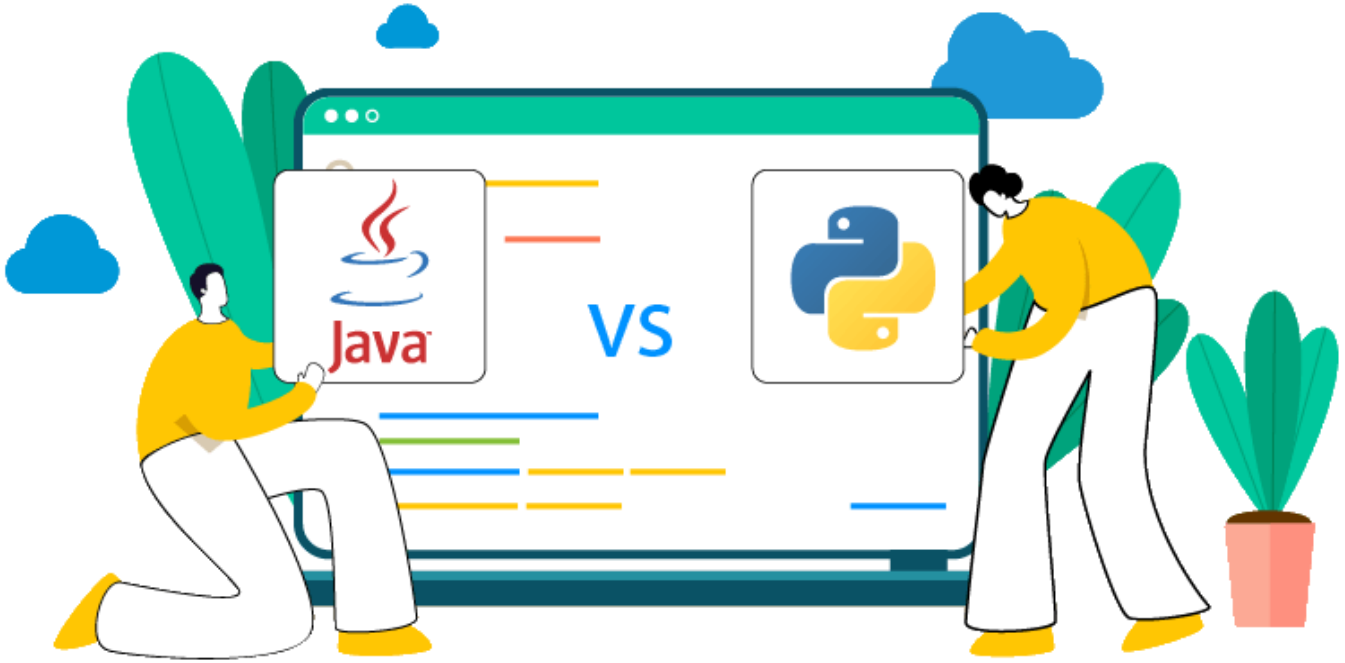
Modifications : A. Tsikhanovich
Auteur : Nicolas Hurtubise , DIRO
Dérivé de documents préparés par Pascal Vincent

AU PROGRAMME …

- Java vs Python
- Programme Java de base
- Types
- Tableaux
- Strings
- Entrées/sorties d'un programme
- Modèle mémoire

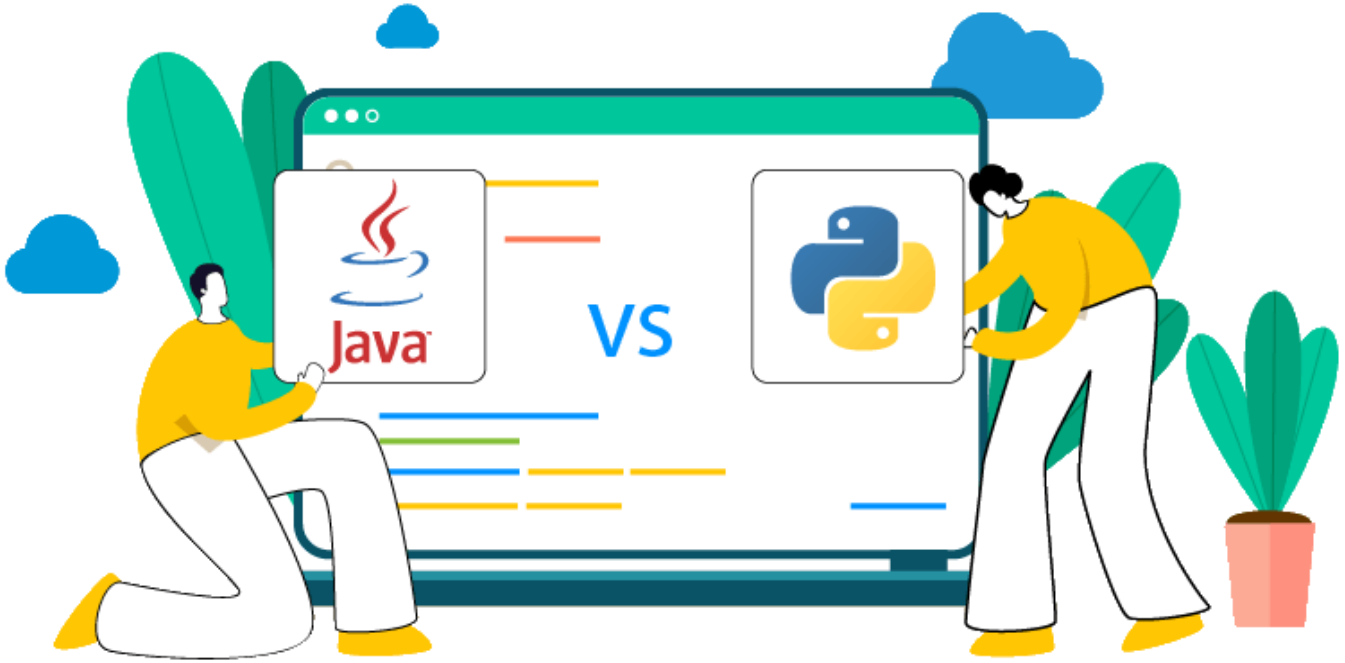
JAVA VS PYTHON

- Python
  - Langage de script, interprété
  - Typage dynamique



https://www.cours-gratuit.com/tutoriel-python/java-vs-python-quel-langage-de-programmation-est-le-meilleur

PYTHON VS JAVA

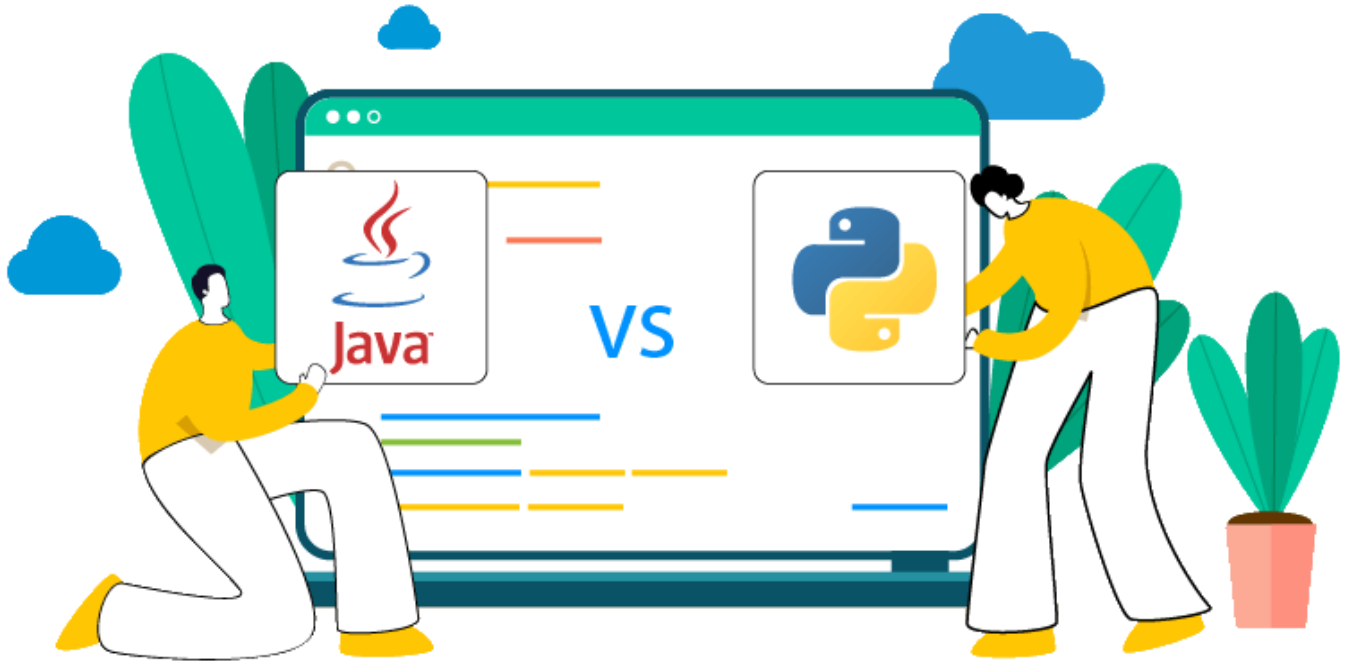


https://www.cours-gratuit.com/tutoriel-python/java-vs-python-quel-langage-de-programmation-est-le-meilleur

- Java
  - Langage compilé (en bytecode)
  - Orienté objet par classes
  - Typage statique
  - Langage plus rigide, ne tolère pas d'erreurs
  - Conçu chez Sun Microsystems pour être "general purpose"

PYTHON VS JAVA

- Java
  - Utilisé un peu partout
    - Applications de bureau
    - Systèmes embarqués
    - Applets sur le web
    - Serveurs web
    - Applications mobiles sur Android



https://www.cours-gratuit.com/tutoriel-python/java-vs-python-quel-langage-de-programmation-est-le-meilleur

HELLO WORLD!

- Programme qui affiche "Hello, World !" à l'écran
- Python

```
>>> print("Hello World!")
    Hello World!
```

HELLO WORLD!

- Programme qui affiche "Hello, World !" à l'écran
- Java

```java
// Fichier: Hello.java
public class Hello {
    public static void main(String[] args) {
        // Votre code ici
        System.out.println("Hello, World !");
    }
}
```

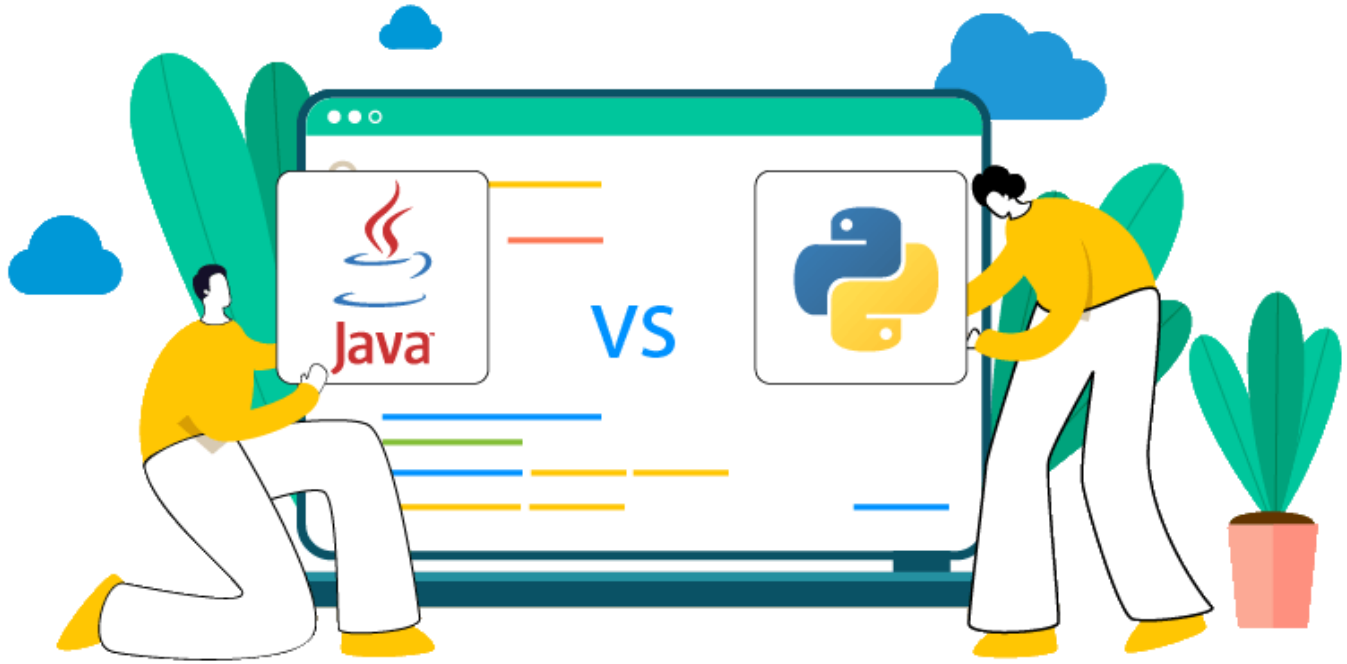
HELLO WORLD!

```java
// Fichier: Hello.java
public class Hello {
    public static void main(String[] args) {
        // Votre code ici
        System.out.println("Hello, World !");
    }
}
```

- Java
  - `main`: la fonction principale de votre programme, celle qui est appelée au début du programme
  - `public static void` ? `String[] args` ?
  - `System.out.println` : fonction équivalente à `print`

EXÉCUTION

- Python
  - Lancer dans codeBoot ou
  - sur la ligne de commande
  - avec un interprète
- Java



https://www.cours-gratuit.com/tutoriel-python/java-vs-python-quel-langage-de-programmation-est-le-meilleur

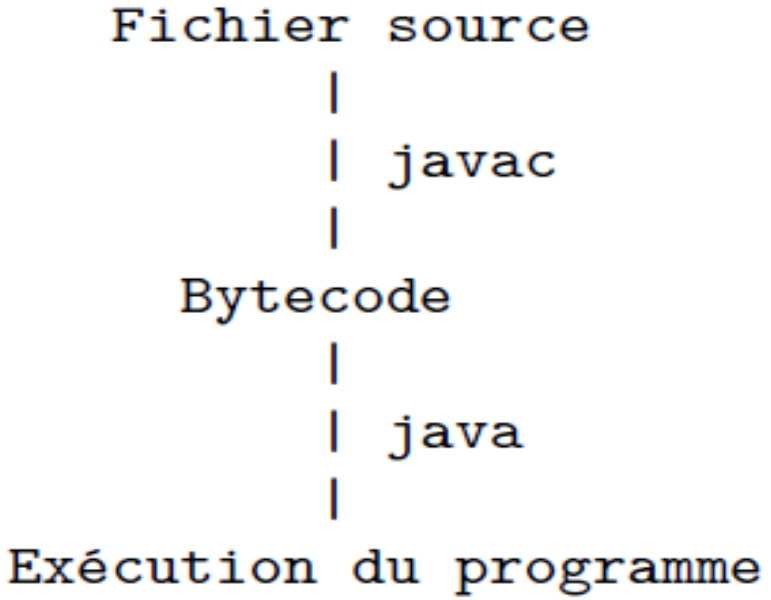
Java

Compilation nécessaire avant d'exécuter :

```
javac Hello.java # crée le fichier "Hello.class"
java Hello # Exécute le code dans Hello.class
```

COMPILATION DE JAVA

- Plutôt que d'exécuter directement le code Java, on le transforme d'abord en *bytecode*
- *Bytecode* == un langage intermédiaire entre le java (langage de programmation) et le binaire (langage machine)
- Accélère l'exécution : on fait une partie du travail une seule fois plutôt qu'à chaque exécution



COMPILATION DE JAVA

```java
// Fichier: Hello.java
public class Hello {
    public static void main(String[] args) {
        // Votre code ici
        System.out.println("Hello, World !");
    }
}
```

```
00 00 00 34 00 1d 0a 00 06 00 0f 09
08 00 12 0a 00 13 00 14 07 00 15 07
06 3c 69 6e 69 74 3e 01 00 03 28 29
43 6f 64 65 01 00 0f 4c 69 6e 65 4e
72 54 61 62 6c 65 01 00 04 6d 61 69
28 5b 4c 6a 61 76 61 2f 6c 61 6e 67
69 6e 67 3b 29 56 01 00 0a 53 6f 75
69 6c 65 01 00 0a 48 65 6c 6c 6f 2e
0c 00 07 00 08 07 00 17 0c 00 18 00
48 65 6c 6c 6f 2c 20 77 6f 72 6c 64
0000240 20 21 07 00 1a 0c 00 1b 00 1c 01 00 05 48 65 6c
0000260 6c 6f 01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 4f
0000300 62 6a 65 63 74 01 00 10 6a 61 76 61 2f 6c 61 6e
0000320 67 2f 53 79 73 74 65 6d 01 00 03 6f 75 74 01 00
...
```

PYTHON, LANGAGE DYNAMIQUEMENT TYPÉ

- En Python, une variable peut prendre n'importe quelle valeur

```
>>> a = "Hello World!"
>>> a = 5
>>> a = True
```

JAVA, LANGAGE STATIQUEMENT TYPÉ

- Java : une variable doit toujours contenir le même type de donnée
- Spécifié à la déclaration

```
// On déclare une variable en précisant son type
// int == integer == nombre entier
int a = 0;

// Invalide, a peut seulement contenir un entier
a = "Bonjour !";

String b = "abc";

// Pas valide non plus...
b = true;
```

AU PROGRAMME …

- Java vs Python
- Programme Java de base
- **Types**
- Tableaux
- Strings
- Entrées/sorties d'un programme
- Modèle mémoire

TYPES

- **Type** définit la nature des valeurs que peut prendre une donnée, ainsi que les opérateurs qui peuvent lui être appliqués

- Si on connaît d'avance quel type de donnée une variable contient, on peut s'en servir pour :
  - Optimiser le code
  - Utiliser moins d'espace en mémoire
  - Prévenir des erreurs

TYPES DE VARIABLES, JAVA

Il y a quelques **types primitifs** :

- int : integer, nombre entier 32 bits
- float : nombre à virgule flottante sur 32 bits
- long : "long" nombre entier, sur 64 bits
- double : nombre à virgule flottante *double précision*, sur 64 bits
- boolean : valeur de vérité true/false
- char : caractère textuel (un seul)
- byte : nombre entier 8 bits
- short : "short int", nombre entier 16 bits

TYPES DE VARIABLES, JAVA

Par défaut, les nombres entiers littéraux sont des `int` et les nombres décimaux littéraux sont des `double`

```java
int age = 25;
double nombreDecimal = 123456789.0;


// Suffixe L pour spécifier un "long int"
long grandNombreEntier = 12345678910L;

// Suffixe f pour spécifier un float 32 bits
float nombreFlottant32bits = 15.3f;

boolean isCoffeeCold = false; // ou true
char caractere = 'a'; // Notez les *guillemets simples ici*
```

TYPES DE VARIABLES, JAVA

Important : le résultat des opérations dépend du type des variables

```java
double a = 3.0;
double b = 2.0;

System.out.println(a + b); // Affiche 5.0 (double)



System.out.println(a / b); // Affiche 1.5 (double)
```

TYPES DE VARIABLES, JAVA

Important : le résultat des opérations dépend du type des variables

```java
int a = 3;
int b = 2;

System.out.println(a + b); // Affiche 5 (int)

/* Un calcul effectué sur deux int aura
   pour résultat un int, attention aux divisions */
System.out.println(a / b); // Affiche 1 ! (int)
```

TYPES DE VARIABLES, JAVA

Solution au problème : on peut forcer à changer une valeur de type avec un **cast**

```java
int a = 3;
int b = 2;

// (double) a => 3.0
// (double) b => 2.0
System.out.println((double) a / (double) b);
// Affiche 1.5
```

TYPES DE VARIABLES, JAVA

- À noter : les variables sont automatiquement considérées comme leur équivalent "plus large" au besoin :
  `byte` → `short` → `int` → `long` → `float` → `double`

```java
int a = 3;

/* Convertit automatiquement a en double
   pour faire le calcul */
System.out.println(a / 2.0); // Affiche 1.5
```

TYPES DE VARIABLES, JAVA

Pour forcer un type "plus spécifique", on doit utiliser un (cast)

```java
double c = 3.0;
double d = 2.0;

/* Affiche 1.5, car a est un double,
   donc le calcul se fait en doubles */
System.out.println(c / 2);

/* Affiche 1, car les deux opérandes
   de la division sont des entiers */
System.out.println((int) c / (int) d);
```

TYPES DE VARIABLES, JAVA

Notez : un cast de `float` en `int` tronque la partie fractionnaire (ça n'arrondit pas)

```
System.out.println((int) 4.6); // Affiche 4
System.out.println((int) -4.6); // Affiche -4
```

RÉSUMÉ SUR LES TYPES

- Toute variable a un type fixe défini à la déclaration
- Les types primitifs les plus communs sont :

Type	Exemple de valeur littérale
byte	(byte) 99
short	(short) 1024
int	65535
long	100000000000L
float	3.5f
double	123.456, 1.23456e2, 123456e-3
char	'a', '?', '.', ...
boolean	true ou false

RÉSUMÉ SUR LES TYPES

- La conversion de types numériques se fait implicitement au besoin, *tant que ça ne risque pas de faire perdre de la précision au programme*
- Convertir un nombre entier plus petit vers un nombre entier plus grand
  - Ex.: `byte` → `int`, `int` → `long`, ...
- Convertir un nombre flottant plus petit en un nombre flottant plus grand
  - Ex.: `float` → `double`
- Convertir un nombre entier en un nombre flottant
  - Ex.: `int` → `float`, `int` → `double`, `long` → `float`, ...

```
int a = 10;
float b = a; // OK: conversion implicite int->float
float c = 100; // OK: int -> float
```

RÉSUMÉ SUR LES TYPES

- Quand il y a un risque de perdre de la précision, on **doit** caster explicitement la valeur
- C'est une façon de dire au compilateur :

  Ne t'inquiète pas, je sais ce que je fais
  Si on perd de la précision, c'est mon problème

```
int x = 2500;
short y = (short) x;

// INCORRECT: car 1.5 représente un double : précision perdue
float z = 1.5;

float z = 1.5f; // OK: littéral float
```

Lire le document *Complément : Conversions entre types de base (Pascal Vincent)* pour plus d'exemples et de précisions

AU PROGRAMME …

- Java vs Python
- Programme Java de base
- Types
- Fonctions
- Tableaux
- Strings
- Entrées/sorties d'un programme
- Modèle mémoire

FONCTIONS

- En Java, toute instruction doit se trouver dans une fonction
- La première fonction appelée est la fonction `main`, qui est déclarée avec :

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

- `public static` : pour l'instant, utilisez ça tel quel sans demander ce que ça mange en hiver, ça sera expliqué plus tard

FONCTIONS

- Les valeurs passées en paramètres à une fonction et la valeur qu'une fonction retourne sont également typées
- Lorsqu'on déclare une fonction, on doit donc inclure des informations sur les types (paramètres et valeur de retour)

Java

```java
public static int carre(int x) {
    return x * x;
}
```

FONCTIONS

Les fonctions qui ne retournent pas de valeur doivent avoir pour type de retour `void`

Java

```java
public static void direBonjour() {
    System.out.println("Bonjour !");
}
```

FONCTIONS

- La **signature d'une fonction** est définie en Java comme étant son nom + le type de ses arguments

Java

```
public static int carre(int x) {
    return x * x;
}
```

Signature

```
carre(int)
```

NOTE SUR LA PORTÉE DES VARIABLES

```java
public static double serieHarmonique(int n) {
    double somme = 0;

    for(int i=1; i<=n; i++) {
        double terme = 1.0/i;
        somme += terme;
    }

    System.out.println(i); // Invalide !
    System.out.println(terme); // Invalide !

    return somme; // OK
}
```

- **Portée d'une variable** est la partie du programme où la variable est accessible

- Variables locales
  - En programmation Java, un **bloc** est une portion de code qui est délimitée par des accolades ({})
  - En Java, la portée des variables locales est limité au **bloc** dans lequel les variables ont été déclarées

AU PROGRAMME …

- Java vs Python
- Programme Java de base
- Types
- Fonctions
- Tableaux
- Strings
- Entrées/sorties d'un programme
- Modèle mémoire

TABLEAUX

- En Java, les tableaux, comme les variables, ne peuvent contenir qu'un seul type de donnée
- Si on veut un tableau contenant des entiers, on doit déclarer la variable reliée comme étant de type "tableau de int" : `int[]`

TABLEAUX

- Les tableaux en Java ont une **taille fixe**
- On initialise un tableau en précisant sa taille :

```
int[] a = new int[3]; // Nouveau tableau vide de taille 3
a[0] = 1;
a[1] = 2;
a[2] = 3;

/* Alternativement, on peut initialiser directement avec
   un tableau littéral
   Java comprend qu'on veut un tableau de taille 5 */
int[] b = {1,2,3,4,5};
```

TABLEAUX

On peut connaître le nombre d'éléments dans un tableau en utilisant `tableau.length` :

```
int[] tab = {10,20,30};
System.out.println(tab.length); // Affiche 3
```

TABLEAUX

- Un tableau 2D est un tableau dont tous les éléments sont de type “tableau 1D”

```
int[][] a = new int[2][5];

// a[0] est de type "int[]" (tableau de int)

/* Tableaux 2D littéraux : Java comprend
   qu'on veut un tableau de taille 2 dans lequel
   chaque élément est un tableau de int de taille 3 */
int[][] b = {{1,2,3},{4,5,6}};

int[][] c = {{1},{2,3},{4,5,6}};
```

TABLEAUX

- Comme en Python, la comparaison de deux tableaux via l'opérateur == ne donne pas le résultat voulu, puisqu'on se retrouve à comparer des **références mémoire**
- Java propose une fonction pour tester si deux tableaux contiennent les mêmes valeurs :

```
Arrays.equals(tableau1, tableau2);
```

- **Attention** : ça ne teste que la première dimension. On peut cependant utiliser `Arrays.deepEquals(tableau1, tableau2)` pour tester récursivement l'égalité de tableaux multidimentionnels

TABLEAUX

Attention !

Les tableaux en Java ont une taille fixe, il **n'**existe **pas** de fonctions comme en Python pour ajouter/retirer des éléments

TABLEAUX

Pour ajouter un élément à la fin d'un tableau de taille N, on doit donc :

- Créer un *nouveau tableau* de taille $N + 1$
- Copier les N premiers éléments du tableau original dedans
- Ajouter un élément de plus

TABLEAUX

```java
public static void main(String[] args) {
    int[] t = {10, 20, 30}; // Tableau de 3 cases

    // Création d'un nouveau tableau de 4 cases
    int[] temp = new int[t.length + 1];

    // Copie des éléments existants
    for(int i=0; i<t.length; i++)
        temp[i] = t[i];

    // Ajout d'un élément à la fin
    temp[t.length] = 40;

    t = temp; // t référence maintenant le nouveau tableau à 4 cases
}
```

TABLEAUX

Même principe si on souhaite retirer un élément du tableau :

- Créer un *nouveau tableau* de taille N - 1
- Copier les N - 1 éléments à conserver dans le nouveau tableau

TABLEAUX

Pas très pratique...

Solutions :

- Connaître d'avance la taille de nos tableaux
- Profiter de l'orienté objet pour définir des structures plus complexes... On y reviendra plus tard

AU PROGRAMME …

- Java vs Python
- Programme Java de base
- Types
- Fonctions
- Tableaux
- Strings
- Entrées/sorties d'un programme
- Modèle mémoire

STRING

Rappel

Un ordinateur ne peut que stocker que des nombres. Stocker du texte demande donc de :

1. Convertir chaque caractère en chiffre (ex.: via la table ASCII)
2. Stocker une suite de chiffres pour former un texte complet

STRING

- "Chaîne de caractères" == Tableau de `char`

Java

```
char[] cours = {'I', 'F', 'T', '1', '0', '2', '5'};
```

STRING

- Java propose le type spécial `String` pour manipuler des chaînes de caractères
- On peut utiliser les doubles guillemets pour créer une `String` littérale :

```
String nom = "Jimmy Whooper";
String chansonFavorite = "Ces Gens Qui Dansent";
```

STRING

Comme en Python, l'opérateur + sert autant à l'addition de nombres qu'à la concaténation de Strings

```
String phrase = "Bonjour mon ami.";

System.out.println(phrase + " Comment vas-tu ?");
// => "Bonjour mon ami. Comment vas-tu ?"

System.out.println("10" + "20");
// => "1020", car on concatène des String
```

STRING

- Du moment qu'au moins un des opérandes du + est de type `String`, l'autre est converti en `String` au besoin et l'opération effectuée est une concaténation (mise bout-à-bout)
- La valeur résultante est également de type `String`

```
System.out.println(25 + "10"); // affiche "2510"

int b = 123;
System.out.println("a" + b); // affiche "a123"
```

STRING

- On utilise la méthode `.length()` qui retourne le nombre de caractères d'une `String` sous la forme d'un int
- Notez qu'un espace compte aussi pour 1 caractère

```
"Allo".length() // vaut 4

String phrase = "Bonjour mon ami.";

phrase.length() // vaut 16

(" 25"+10).length() // vaut 5
```

STRING

- Une `String` est un type qui représente une chaîne de caractère, c.a.d. une séquence de caractères (`char`)
- Les caractères ont une position (ou index) numérotée à partir de 0

Exemple: `"Nom"` est une chaîne formée de 3 caractères (donc de longueur 3) :

```
à la position 0 c'est 'N'
à la position 1 c'est 'o'
à la position 2 c'est 'm'
```

STRING

- Pour obtenir à partir d'une `String` s, le caractère (char) à une position (ou index) donnée i il suffit d'appeler s.charAt(i)

```
"Allo".charAt(0) // vaut 'A', de type char
// (et non pas "A" qui serait de type String)

String n = "123";
n.charAt(n.length()-1) // vaut '3' (et non pas "3" ni 3)
n.charAt(3) // donnera lieu à une erreur à l'exécution
(n + 0).charAt(3) // vaut '0' (et non pas "0", ni 0)
```

STRING

Attention : on ne peut pas ainsi modifier un caractère dans une `String` :

```
n.charAt(1) = 'Z'; // donnera une erreur à la compilation
```

STRING

- Pour extraire une partie d'une `String` s on fait appel à la méthode `s.substring(debut, fin)`
- Le résultat de cette expression est une sous-chaîne de s :
  - Commençant au caractère à la position debut
  - S'arrêtant juste avant le caractère à la position `fin`
  - La longueur de la sous-chaîne résultante est donc `fin - debut`
  - Ce résultat est de type `String`
  - Il s'agit d'une nouvelle `String`

STRING

Exemples :

```
"Bonjour".substring(1,6) // vaut "onjou" (pas "Bonjour" ni "onjour")

String salut = "Allo";
int pos = 0;

salut.substring(pos, pos) // vaut "" (et non pas "A")
salut.substring(pos, pos+1) // vaut "A" (et non pas 'A')
salut.substring(pos, pos+2) // vaut "Al"
salut.substring(0, salut.length()-1) // vaut "All" (pas "Allo")
salut.substring(salut.length()-1, salut.length()) // vaut "o"
```

STRING

- Pour obtenir une `String` où le caractère à la positions i a été remplacé par 'X' on peut écrire par exemple:

```
String salut = "Allo";

int i = 1;

String salut2 = salut.substring(0,i) +
      'X' + salut.substring(i+1,salut.length());
// salut2 aura alors la valeur "AXlo".
```

STRING

- La méthode `.toUpperCase()` retourne une version où toutes les minuscules ont été transformées en majuscules

```
"BonJour!".toUpperCase() // vaut "BONJOUR!"
```

- La méthode `.toLowerCase()` retourne une version où toutes les majuscules ont été transformées en minuscules

```
"BonJour!".toLowerCase() vaut "bonjour!"
```

STRING

- *Attention !* Une string n'est pas un type primitif
- C'est un type spécial
- Contrairement à en JavaScript, on **ne** peut **pas** comparer deux `Strings` avec `==` :

Java

```
String a = "gazoline";
String b = "gazoline";

System.out.println(a == b); // Affiche false
```

Puisqu'il ne s'agit pas d'un type primitif, on compare des *références mémoire*, similairement aux tableaux

STRING

- Cette approche est un piège commun lorsqu'on commence à programmer en Java, mais est cohérente avec ce qui se passe réellement en mémoire

- Pour comparer deux strings, on doit utiliser la méthode `.equals()` :

```
String a = "gazoline";
String b = "gazoline";

System.out.println(a.equals(b));
// => true

System.out.println(("gaz" + "oline").equals("gazoline"));
// => true
```

STRING

- Notez : lors de la comparaison, la casse est importante :

```
System.out.println("abc".equals("ABC"));
// => false

System.out.println("aBc".equals("abc"));
// => false
```

STRING

- Si les différences majuscules/minuscules ne nous importent pas, on peut comparer avec `.equalsIgnoreCase()`

```
System.out.println("abc".equalsIgnoreCase("ABC"));
// => true

System.out.println("aBc".equalsIgnoreCase("abc"));
// => true
```

- Aka, savoir laquelle est avant l'autre dans le dicitonnaire

STRING

- Deux chaînes de caractères peuvent être identiques ou différentes
- Si elles sont différentes, on peut aussi dire qu'une est "supérieure" ou "inférieure" à l'autre selon qu'on la placerait après ou avant dans un dictionnaire
- Cette relation d'ordre définie entre les `String` s'appelle l'*ordre lexicographique*
- Puisque les `String`s ne sont pas des types primitifs, on ne peut pas utiliser les opérateurs de comparaison `==`, ni `>` ou `<` pour les comparer

STRING

- On vient de voir la méthode .equals() qui évalue si deux `String` sont identiques
- La méthode `.compareTo()` permet en plus de savoir laquelle précède l'autre dans l'ordre lexicographique

STRING

- `str1.compareTo(str2)` retourne un `int` dont il suffit de considérer le signe pour savoir quelle `String` précède l'atre dans le dicitonnaire.

```
"ABC".compareTo("ABC")
// vaut 0 car les deux String sont identiques.

"ABC".compareTo("ABZ")
// donne un entier négatif car "ABC" viendrait avant "ABZ"
// dans un dictionnaire ("ABC" est considéré plus petit que "ABZ")

"ABZ".compareTo("ABC")
// donne un entier positif car "ABZ" viendrait après "ABC"
// dans un dictionnaire ("ABZ" est considéré plus grand que "ABC")

"ABZ".compareTo("ABCDEFGH")
// donne un entier positif car"ABZ" est considéré plus
// grand que "ABCDEFGH" dans l'ordre lexicographique
```

STRING

- La comparaison "lexicographique" se fait via la valeur numérique du code Unicode qui représente les caractères
- Les chiffres sont avant les lettres majuscules qui sont avant les lettres minuscules

```
   30 40 50 60 70 80 90 100 110 120
  ------------------------------
0:    (  2  <  F  P  Z  d   n   x
1:    )  3  =  G  Q  [  e   o   y
2:    *  4  >  H  R  \  f   p   z
3: !  +  5  ?  I  S  ]  g   q   {
4: "  ,  6  @  J  T  ^  h   r   |
5: #  -  7  A  K  U  _  i   s   }
6: $  .  8  B  L  V  `  j   t   ~
7: %  /  9  C  M  W  a  k   u  DEL
8: &  0  :  D  N  X  b  l   v
9: '  1  ;  E  O  Y  c  m   w
```

- Donc "123".compareTo("ABC") donne un entier **négatif**
  - Car les chiffres viennent avant les lettres majuscules
- Donc "ABc".compareTo("ABZ") donnera un entier **positif**
  - Car les lettres minuscules viennent *après* les lettres majuscules dans la table Unicode
  - Donc 'c' > 'Z' (même si ça serait l'inverse dans le dictionnaire)

STRING

- Si on veut ignorer les différences majuscule/minuscule, on peut utiliser la méthode `.compareToIgnoreCase()`, ce qui revient au même que de tout mettre en minuscules et ensuite appeler `.compareTo()`

STRING

Exemple: "Bonjour Monsieur!" contient "jour" à la position 3
On peut pour obtenir ce résultat, en utilisant `.indexOf()` :

```
"Bonjour Monsieur!".indexOf("jour") // vaut 3
```

STRING

- si la chaîne ne contient pas la sous-chaîne recherchée, indexOf retourne -1

```
"Bonjour Monsieur!".indexOf("jours") // vaut -1
```

STRING

- On peut optionnellement, comme second paramètre explicite à `.indexOf()`, indiquer la position à partir de laquelle chercher

```
"Bonjour Monsieur!".indexOf("on", 0) // vaut 1
// Remarquez qu'il s'agit de la position du premier "on" trouvé

"Bonjour Monsieur!".indexOf("on", 5) // vaut 9
// Le premier "on" trouvé à partir de la position 5

"Bonjour Monsieur!".indexOf("on", 10) // vaut -1
// "on" introuvable passé la position 10
```

STRING

- Note : on peut aussi utiliser un char à la place d'une `String` pour ce qui est recherché (mais l'entité sur laquelle on fait la recherche doit être une `String`, pas un char)

```
"Bonjour".indexOf('j') // équivalent à "Bonjour".indexOf("j")
```

STRING

- Rappel: Pour convertir entre des types primitifs de Java on peut souvent utiliser un cast

```
(int) 3.25 // vaut 3 et ce résultat est de type int
```

- *Mais...* les `String` ne sont pas des types simples de Java, on ne peut pas utiliser de cast pour convertir une `String` en `int` ou un `double` en sa représentation `String`

```
(int) "3.25" // INCORRECT
```

STRING

- Pour convertir une valeur d'un type primitif (`char`, `int`, `long`, `double`, `float`, `boolean`, ...) en `String`, deux possibilités :

```
10 + "" // vaut "10"

128.7 + "" // vaut "128.7" et est de type String

String.valueOf(10) // vaut "10"

boolean a = true;
a + "" // vaut "true"
String.valueOf(a) // vaut "true"
```

STRING

- Pour convertir une `String` représentant un nombre en un type primitif de Java, on peut utiliser des fonctions prédéfinies :

```
Integer.parseInt("1" + "2") // vaut 12 et est de type int
Double.parseDouble("-3e-1") // vaut -0.3 et est de type double

Integer.parseInt("bonjour") // lance une erreur à l'exécution

/* Notez, des méthodes similaires existent pour
   les autres types primitifs float, long, ... */
```

DIFFÉRENCES SUR LA TAILLE ARRAY ET STRING

Notez bien la différence pour trouver la taille :

- Array: `tab.length`
- String: `str.length()`

Différences sur la taille Array vs String

ENTRÉES ET SORTIES

Affichage à l'écran

- System.out.println
- System.out.print (sans saut de ligne)

Interaction avec l'utilisateur

- `args` : arguments de la ligne de commande
- `Scanner` : lecture de données interactive

ENTRÉES ET SORTIES

Arguments passés via la ligne de commande

```
public class Hello {
    public static void main(String[] args) {
        // Votre code ici
        System.out.println("Hello, " + args[0] + " !");
    }
}
```

- **args**: tableau de **Strings** passé en paramètre lorsqu'on lance le programme via la ligne de commande

```
$ javac Hello.java
$ java Hello monargument
Hello, monargument !
```

ENTRÉES ET SORTIES

```java
public class Max {

    public static void main(String[] args) {
        int max = -1;

        for(int i=0; i<args.length; i++) {
            max = Math.max(max, Integer.parseInt(args[i]));
        }

        System.out.println("Maximum=" + max);
    }
}
/* Exécution :
$ javac Max.java
$ java Max 1 6 3 1 5 3
Maximum=6
*/
```

ENTRÉES ET SORTIES

Lecture de données interactive

- Scanner : permet de lire interactivement des données sur la ligne de commande

```
public static void main(String[] args) {
    java.util.Scanner scanner =
        new java.util.Scanner(System.in);

    int number = 0;

    while(number >= 0) {
        number = scanner.nextInt();
        System.out.println("Nombre entré : " + number);
    }
}
```

Lecture de données interactive

- On peut importer `java.util.Scanner` pour pouvoir écrire directement `Scanner` dans le code

```
import java.util.Scanner;

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    int number = 0;

    while(number >= 0) {
        number = scanner.nextInt();
        System.out.println("Nombre entré : " + number);
    }
}
```

ENTRÉES ET SORTIES

ENTRÉES ET SORTIES

- `Scanner` permet de lire plusieurs types de données depuis la console :
  - `nextInt()` pour lire un `int`
  - `nextDouble()` pour lire un `double`
  - `nextByte()`, `nextFloat()`, ...
  - `nextLine()` une `String` allant jusqu'à la fin de la ligne de texte (donc jusqu'au prochain `\n`)

MODÈLE MÉMOIRE

- Conceptuellement, les données de programme sont stockées dans les deux zones de la mémoire principale : Pile et Tas

  - La *pile* (en anglais, *stack*)
  - Le *tas* (en anglais, *heap*)

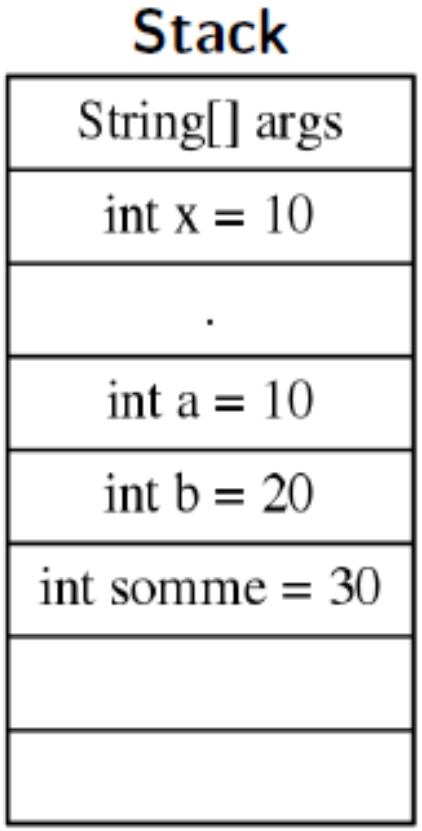
  Les variables définies dans les fonctions appelées sont allouées sur la pile, tandis que les données plus complexes sont allouées dans le tas

MODÈLE MÉMOIRE

```java
public static void main(String[] args) {
    int x = 10;
    // -> ici <-
    System.out.println(add(x, 20));

}

public static int add(int a, int b) {
    int somme = a + b;

    return somme;
}
```

MODÈLE MÉMOIRE

```java
public static void main(String[] args) {
    int x = 10;

    System.out.println(add(x, 20));

}

public static int add(int a, int b) {
    int somme = a + b;
    // -> ici <-
    return somme;
}
```

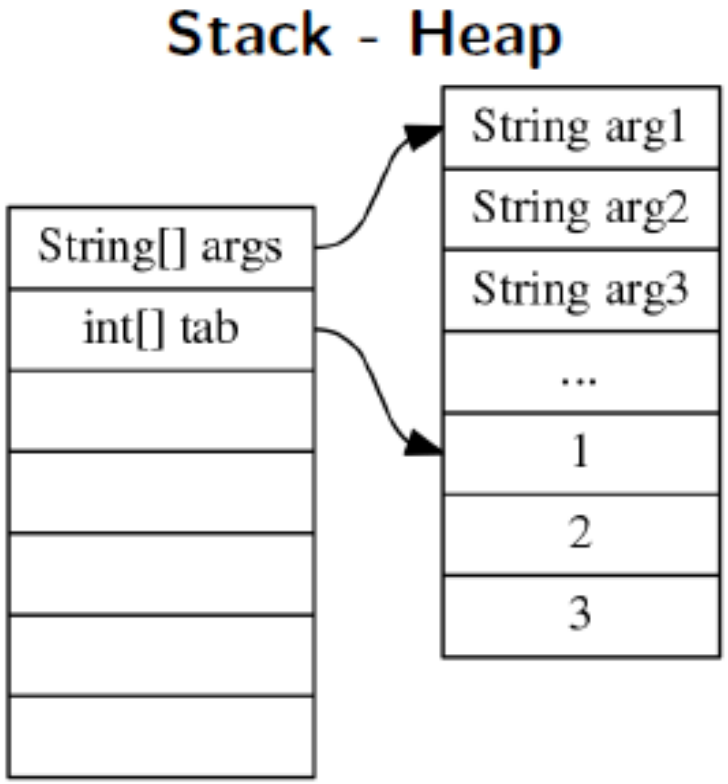


MODÈLE MÉMOIRE

```java
public static void main(String[] args) {
    int x = 10;

    System.out.println(add(x, 20));
    // -> ici <-
}

public static int add(int a, int b) {
    int somme = a + b;

    return somme;
}
```

MODÈLE MÉMOIRE

```
public static void main(String[] args) {
    int[] t = {1,2,3};
    // -> ici <-
    System.out.println(moyenne(t));
}

...
```



MODÈLE MÉMOIRE

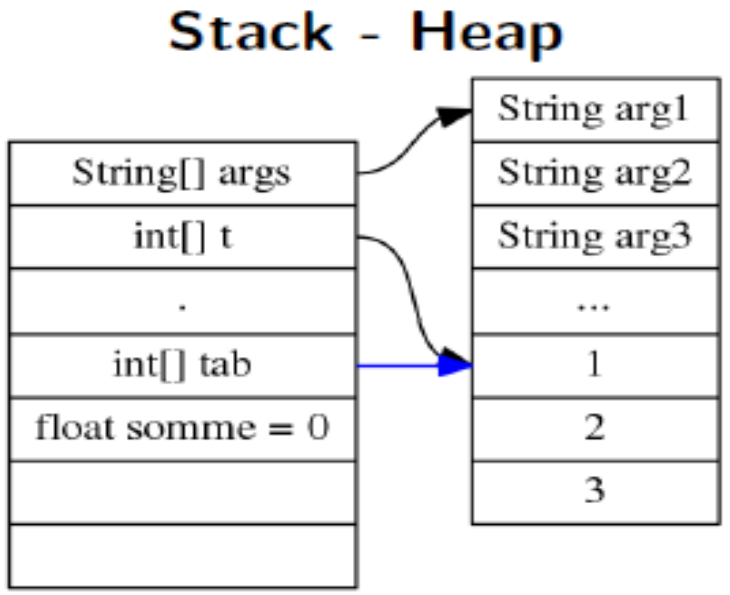
```java
public static void main(String[] args) {
    int[] t = {1,2,3};

    System.out.println(moyenne(t));
}

public static float moyenne(int[] tab) {

    float somme = 0;
    // -> ici <-

    for (int i = 0; i < tab.length; i++) {
        somme += tab[i];
    }

    return somme / tab.length;
}
```

Stack

String[] args
int[] t
.
int[] tab
float somme = 0

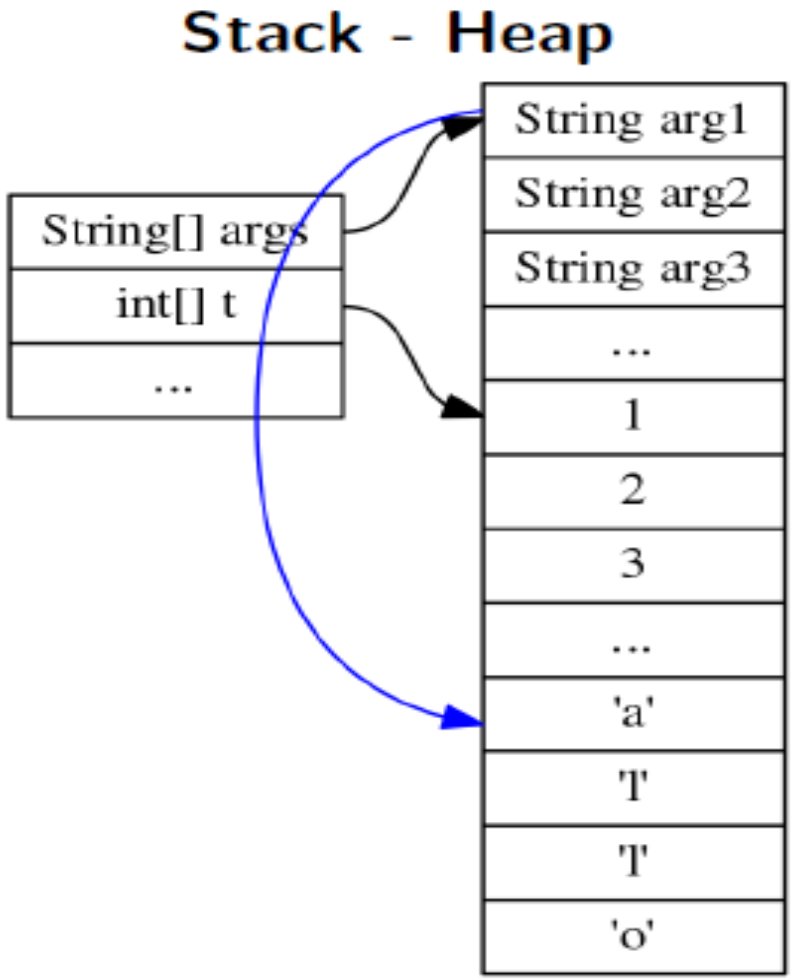
MODÈLE MÉMOIRE

```java
public static void main(String[] args) {
    int[] t = {1,2,3};

    System.out.println(moyenne(t));
}

public static float moyenne(int[] tab) {

    float somme = 0;
    // -> ici <-

    for (int i = 0; i < tab.length; i++) {
        somme += tab[i];
    }

    return somme / tab.length;
}
```



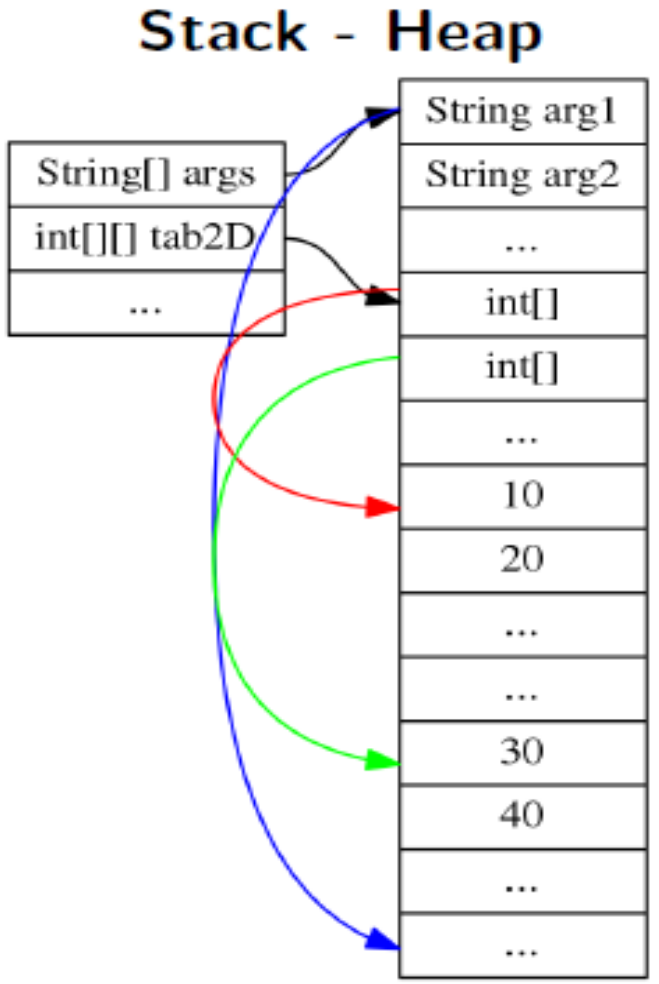
MODÈLE MÉMOIRE

- Le diagramme précédent n'est pas tout à fait juste...
- Comme une `String` n'est pas un type primitif, une case qui contient un `String` est en fait une *référence* vers un autre bout de la mémoire qui contient les informations sur la `String` (le tableau de `char` sous-jacent, la longueur, ...)



MODÈLE MÉMOIRE

```
public static void main(String[] args) {
    int[][] tab2D = new int[2][2];
    tab2D[0][0] = 10;
    tab2D[0][1] = 20;

    tab2D[1][0] = 30;
    tab2D[1][1] = 40;

    System.out.println(moyenne(tab2D[0]));
    // => 15.0

    System.out.println(moyenne(tab2D[1]));
    // => 35.0
}

public static float moyenne(int[] tab) {
    ...
}
```



MODÈLE MÉMOIRE

En résumé :

- Les cases mémoire ne peuvent contenir que des *types primitifs* (`int`, `char`, `double`, ...) ou des *références* vers d'autres endroits en mémoire
- Quand on appelle une fonction, les valeurs passées en paramètres sont *copiées* dans d'autres cases mémoire
- Dans le cas de types complexes, c'est la *référence* qui est copiée

POURQUOI JAVA ?

Java peut sembler plus lourd à première vue :

- Besoin de compiler les programmes avant de les exécuter
- Typage statique des variables
- Tableaux de taille fixe
- `public static void main(String args[])` ...