

IFT1025 Programmation 2

De Python à Java

Franz Girardin

3 Mai 2023

2 | CHAPITRE 1 Introduction

- 1.1 Particularité de Java et Python 2

3 | CHAPITRE 2 Notions Essentielles

- 2.1 Types Java 3
- 2.2 Fonctions Java 6
- 2.3 Tableaux Java 7
- 2.4 Chaîne de caractères *string* 8
- 2.5 Entrées et Sorties 13

1.1 Particularité de Java et Python

Concept. Langage statique et langage dynamique

Java est un langage statique ; la vérification du code est effectuée *lors de la compilation*. **Python** est un langage dynamique et la vérification est donc en cours *tout au long de l'exécution de programme*.

Définition Vérification de code

Il s'agit du processus de vérification et d'application des contraintes de types en fonction de la syntaxe permise par le langage.

TABLE 1.1 – Comparaison de Python et Java

Python	Java
<i>Langage interprété</i>	<i>Langage compilé</i>
<i>Typage dynamique</i> et faible quantité de type	<i>Typage statique</i> grande variété de types
<i>Langage tolérant et peu rigide</i>	<i>Langage rigide qui permet peu d'erreurs</i>
<i>Développement plus simple mais exécution plus lente</i>	<i>Développement plus complexe et exécution plus rapide</i>

Exemple. Hello world ! en Python

```
1 print("Hello world!")
```

Exemple. Hello world ! en Java

```
1 public class Example {  
2     public static void main(String[] args) {  
3         System.out.println("Hello world!");  
4     }  
5 }
```

Concept. Compilation de Java

Le code Java est d'abord transformé en bytecode—un langage intermédiaire.

Java est un langage de programmation orienté objet. Le code qu'on écrit est donc parti d'une classe ; la classe principale—`main` dans ce cas-ci. La classe `main` a une méthode : `public static void main (String[] args) {...}`. Le programme qu'on écrit est généralement compris entre les crochets de la méthode `main`

CHAPITRE 2

NOTIONS ESSENTIELLES

2.1 Types Java

Définition Types

Un type définit la nature des valeur que peut prendre une donnée et les opérateurs qui peuvent lui être appliqués.

Note :

Les programmes qui explicitent les types optimisent le traitement et la vitesse d'exécution du code, utilisent moins d'espace mémoire et tendent à engendrer moins d'erreur.

TABLE 2.1 – Types primitifs Java

Type	Description
short	Entier encodé sur 16 bits
int	Entier encodé sur 32 bits
long	Entier encodé sur 64 bits
float	Nombre à virgule encodé sur 32 bits
double	Nombre à virgule encodé sur 64 bits
boolean	Valeur de vérité <code>true</code> ou <code>false</code>
char	Caractère textuel unitque
byte	Nombre entier de 8 bits

Remarque.

Le type d'une variable est précisé lors de la déclaration. Après la déclaration, le type est immuable.

Exemple. Déclaration de variable en Java

```
1 // On declare un variable en precisant son type
2 int a = 0;
3
4 // Formule invalide puisque a precedemment definit ne peut contenir que des entiers
5 a = "Bonjour !";
6
7 String b = "abc";
8
9 // Formule invalide puisque b precedemment definit ne peut etre un boolean
10 b = true;
```

Exemple. Déclaration de variable en Python

```
1 # Toutes ces operations sont permises
2 a = "Hello World!"
3 a = 5
4 a = True
5 b = False
```

Remarque. Entiers et décimaux

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

Exemple.

```
1 int age = 25;
2 double nombreDecimal = 123456789.0;
3
4 // On utilise le suffixe L pour specifier un "long int"
5 long grandNombreEntier = 12345678910L
6
7 // On utilise le suffixe f pour specifier un float de 32 bits
8 float nombreFlottant32bits = 15.3f;
9
10 boolean isCoffeeCold = false;
11
12 // On utilise le guillemet simple pour specifier un char (caractere textuel unique)
13 char caractere = 'a';
```

Note :

Le résultat des opérations dépend du types des variables; le résultat peut être arrondis selon le **type des opérandes**

Exemple. Arithmétique selon le type

```
1 // Addition et division de deux double
2 double a = 3.0;
3 double b = 2.0;
4
5 System.out.println(a+b); //Affiche 5.0 (un double).
6
7 System.out.print(a/b); // Affiche 1.5 (un double)
```

```

1 // Addition et division de deux int
2 int a = 3;
3 int b = 2;
4
5 System.out.println(a + b) // Affiche 5 (un int)
6
7 System.out.println(a/b); // Affiche 1 (un int)

```

Syntaxe. Changer le type grâce à un cast

On peut *forcer le changement d'un type* pour effectuer une opération arithmétique en utilisant un *cast*. On précise entre parenthèse le type désiré suivi de la variable à modifier.

Exemple.

```

1 int a = 3;
2 int b = 2;
3
4 //Affiche 1.5
5 System.out.println(double) a / (double) b);

```

Note :

Les entiers sont automatiquement convertis vers leur équivalent le plus large si nécessaire
 byte -> short -> int -> long -> float -> double

Exemple. Conversion automatique d'un int

```

1 int a = 3;
2
3 //, L'entier sera converti en double pour effectuer l'opération
4 System.out.println(a / 2.0); //Affiche 1.5

```

Syntaxe. Forcer une perte de précision grâce à un cast

On peut forcer un *type plus précis* à devenir moins précis lors d'une opération toujours en utilisant un cast

Exemple.

```

1 double c = 3.0
2 double d = 2.0
3
4 /* Afficherait normalement 1.5, car c est un double
5 et le calcul se ferait sur double */
6 System.out.println(c / 2);
7
8 /* Affiche 1 car les deux opérandes de
9 la division sont des entiers */
10 System.out.println( (int) c / (int) d);

```

Note :

Un cast de `float` en `int` a comme effet de tronquer la partie fractionnaire, plutôt que d'arrondir le résultat.

Exemple.

```
1 System.out.println((int) 4.6); // Affichera 4
2 System.out.println((int) -4.6);
3 // Affichera -4 (aurait affiche -5 si cast arrondissait)
4
```

2.2 Fonctions Java

Concept. Fonction principale

La première fonction d'un programme Java est `public static void main(String args[])`. Elle contient généralement tout le reste du code.

Lorsqu'on déclare une fonction, on doit spécifier le *type des arguments* qu'elle prends et le *type de la valeur retournée* par la fonction.

Exemple. Fonction de base

```
1
2 // La declaration int avant carre indique que la fonction retourne des entiers
3 public static int carre(int x) {
4     return x * x
5     /* La declaration int entre parenthese et avant le x indique que
6        la fonction accepte des entiers */
7 }
8
```

Définition Signature d'une fonction

Il s'agit de la combinaison de son nom et du type de ses arguments. La fonction de l'exemple précédent a comme signature `carre(int)`.

Concept. Portée d'une variable

Il s'agit de la partie du programme où la variable est accessible. Une variable locale est un variable définit dans une fonction donnée et accessible uniquement à cette fonction *ou les éléments faisant parti du bloc de cette fonction*.

Définition Bloc

Il s'agit d'une portion de code qui est délimitée par des accolades `{...}`

Lorsqu'une fonction ne retourne aucune valeur, on l'indique en écrivant `void` avant le nom de la fonction. Les fonctions qui ne retournent rien sont ne sont pas des fonctions mais plutôt des *procédures*.

Exemple. Fonction sans valeur de retour

```
1 public static void direBonjour () {
2     System.out.println("Bonjour !");
3 }
```

2.3 Tableaux Java

Syntaxe. Création d'un tableau en Java

On ne peut avoir qu'un type par tableau en Java. Lorsqu'on déclare le tableau, il faut *spécifier le type des éléments qu'il contient*. Il faut aussi spécifier la quantité d'éléments et *manuellement initialiser le tableau*.

Exemple. Initialisation manuelle et automatique d'un tableau Java

```
1 // La commande new int permet de creer un tableau vide de taille N
2 int[] a = new int[3];
3
4
5 // On specifie les valeurs a chaque position du tableau
6 a[0] = 1;
7 a[1] = 2;
8 a[3] = 3;
9
10 /* On peut aussi initialiser automatiquement avec un tableau litteral
11 Java comprend qu'on veut un tableau de taille 5 */
12 int[] b = {1,2,3,4,5,6}
13
```

Remarque. Trouver la taille d'un tableau

On peut déterminer la quantité d'éléments présents dans un tableau en utilisant la commande `NomDuTableau.length`

Exemple.

```
1 // Créer un tableau d'entiers contenant trois éléments
2 int[] tab = {10, 20, 30};
3
4 // Affiche 3 (la taille du tableau)
5 System.out.println(tab.length);
```

Syntaxe. Tableau 2D

Il s'agit d'un tableau dans lequel chaque élément est également un tableau à une dimension.

Exemple. Création d'un tableau 2D

```
1 /* Methode manuelle : on cree un tableau a deux dimensions
2 "[2]" ou chaque element est un tableau de taille 3.*/
3 int[][] a = new int[2][5]
4
5 /* Methode automatique : Java comprend qu'on veut creer un
6 tableau de taille 2 dans lequel chaque element est un tableau de taile 3 */
7 int[][] b = {{1,2,3}, {4,5,6}}
```

Note :

On ne peut pas comparer la taille de deux tableaux via l'opérateur `==` ; Java considère qu'on veut comparer deux zones mémoires.

Syntaxe. Comparaison de tableau

`Arrays.equals(tableau1, tableau2)` vérifie si `tableau1` a la même taille que `tableau2`.

`Arrays.deepEquals(tableau1, tableau2)` vérifie récursivement si les tableaux multidimensionnels ont la même taille.

Note :

On ne peut pas ajouter des éléments à un tableau Java ; chaque tableau est de taille fixe.

Syntaxe. Ajouter un élément à un tableau

Dans l'ordre, il faut créer un tableau de taille $N + 1$; copier les N premiers éléments du tableau original ; et ajouter un élément de plus.

Exemple. Ajout d'élément par création d'un nouveau tableau

```
1 public static void main(String[] args) {
2     int[] t {10,20,30}; // Cree un tableau de 3 elements
3
4     // Cree un nouveau tableau de 4 elements
5     int[] temp = new int[t.length + 1];
6
7     //Copie les element existants de t dans temp
8     for(int i=0; i<t.length; i++)
9         temp[i] = t[i];
10
11     // Ajoute l'element desire a la fin du tableau
12     temp[t.length] = 40;
13
14     // Assigne la valeur du tableau temp au tableau t
15     t = temp
```

Note :

Lorsqu'on souhaite retirer un élément d'un tableau, le même principe s'applique : Crée un tableau de taille $N - 1$; copier les $N - 1$ éléments à conserver dans le nouveau tableau ; assigner la valeur du nouveau tableau à l'ancien tableau.

2.4 Chaîne de caractères *string*

Concept. Stockage des string

L'enregistrement d'un string en mémoire implique la conversion dudit caractère en chiffre selon un standard—p. ex. ASCII.

Exemple. Création d'un string Java

```
1 String nom = "Jimmy Whooper";
2 char[] tDeChar = {'J','I','M','M','Y',' ','W','H','O','O','P','E','R'};
```

Note :

L'opérateur `+` sert autant à l'addition de nombres qu'à la concaténation de Strings

Exemple. Concaténation de Strings

```
1 String phrase = "Bonjour mon ami.";
2
3 // Affiche : "Bonjour mon ami. Comment vas-tu ?"
4 System.out.println(phrase + " Comment vas-tu ?");
5
6 // Affiche "1020" ; la concatenation des deux Strings.
7 System.out.println("10" + "20");
```

Note :

Lorsqu'on a des opérandes est de type `String`, l'autre est converti en `String` et l'opération effectuée est une concaténation. La valeur résultante est de type `String`.

Exemple. Concaténation de String à un int

```
1 System.out.println(25 + "10"); //Affiche "2510"
2
3 int b = 123;
4 System.out.println("a" + b); // Affiche "a123"
```

Note :

On utilise la méthode `.length` pour retourner le nombre d'éléments d'une chaîne de caractères. Un espace vaut 1 caractère. Le nombre retourné est de type `int`.

Exemple. Déterminer la taille du chaîne

```
1 "Allo".length() // vaut 4
2
3 String phrase = "Bonjour mon ami";
4 phrase.length() // vaut 16
5
6 (" 25"+10.).length() // vaut 5 (l'espace est compté)
7
```

Note :

Un `String` est une chaîne de caractères `char`. Chaque `char` d'un `String` a un index en fonction de sa position dans le `String`.

Exemple.

```
1 char B = "Nom";
2 // B a le char 'N' en position 0
3 // B a le char 'o' en position 1
4 // B a le char 'm' en position 2
5
```

Note :

On peut obtenir le caractère d'un `String s` à la position `i` grâce à la commande `s.charAt(i)`

Exemple. Obtenir le caractère à une position donnée d'un String

```
1  "Allo".charAt(0) /* vaut 'A' de type char
2  ne vaut pas "A" qui serait de type String */
3
4  String n = "123"
5  n.charAt(n.length()-1) // vaut '3' (et non pas "3")
6  n.charAt(3) // engendre une erreur ; n'a pas de position 3
7  (n + 0).charAt(3) // vaut '0' (et non "0", ni 0)
8
```

Note :

On peut extraire une partie d'un String grâce à la méthode `s.substring(debut, fin)`. Le résultat de cette expression est une sous-chaîne de `s`.

Remarque.

La nouvelle chaîne créée par la commande `s.substring(debut, fin)` comment à la position `debut` et se termine avant la position `fin`. La longueur de la sous-chaîne est donc `fin - debut`

Exemple. Création de sous-chaîne

```
1  //Vaut "onjour"
2  "Bonjour".substring(1,6)
3
4  String salut = "Allo";
5  int pos = 0;
6
7  //Vaut ""
8  salut.substring(pos, pos)
9  salut.sustring(pos, pos+1) // vaut "A" (et non pas 'A')
10 salut.substring(pos, pos+2) //vaut "Al"
11
12 salut.substring(0, salut.length()-1) // vaut "All"
13 salut.substrong(salut.length()-1, salut.length()) // vaut "o"
```

Exemple. Modifier le `char` d'un String à la position `i`

```
1  // Soit i == 1
2  int i = 1;
3
4  String salut2 = salut.substring(0,i) + 'X' +
5  salut.substring(i+1, salut.length());
6  // salut2 vaut "AXlo"
7
```

Note :

La méthode `.toUpperCase()` retourne une version où toutes les minuscules ont été transformées en majuscules. On utilise plutôt `.toLowerCase()` pour obtenir une version où toutes les majuscules sont changées en minuscules.

Exemple. Changement de majuscules et minuscules

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

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

Note :

On ne peut pas comparer deux String via l'opérateur == ; Java considère qu'on veut comparer deux zones mémoires.

Syntaxe. Comparaison de deux String

`StringA.equals(StringB)` permet de comparer la longueur des String A et B. La case est sensible. On utilisera `.equalsIgnoreCase()` si on ne veut pas que la case soit sensible lors de la comparaison.

Exemple. Comparer deux String en Java

```
1 String a = "gazoline"
2 String b = "gazoline"
3
4 System.out.println(a == b) // Affiche false
5 System.out.println(a.equals(b)); //Affiche true
6
7 System.out.println(("gaz" + "oline").equals("gazoline"))
8 // Affiche true
9
10 System.out.println("abc".equals("ABC"));
11 // Affiche false ; la case est importante
12 System.out.println("aBc".equals("abc"));
13 // Affiche false
14
15
16 System.out.println("abc".equalsIgnoreCase("ABC"));
17 // Affiche true
18 System.out.println("aBc".equalsIgnoreCase("abc"));
19 // Affiche true
20
```

Définition Ordre lexicographique

Il s'agit de l'ordre dans lequel on place un String par rapport à un autre, en fonction de sa valeur—sachant que chaque char qui compose le string a une valeur déterminée par l'ordre alphabétique.

Note :

On peut utiliser la méthode `compareTo()` pour déterminer quel String précède l'autre.

Exemple. Comparaison de l'ordre lexicographique

```
1 "ABC".compareTo("ABC")
2 // Vaut 0 puisque les deux String sont identiques
3
4 "ABC".compareTo("ABZ")
5 // A une valeur negative puisque C precede Z.
6
7 "ABZ".compareTo("ABC")
8 // Vaut un entier positif, puisque Z > C
9
10 "ABC".compareTo("ABCDEFGH")
11 // Vaut un entier positif puisque "ABZ" est considere
```

12 comme plus grand que "ABCDEFGH"

Note :

La comparaison lexicographique se fait avec la valeur numérique du code Unicode du caractère correspondant. L'ordre de priorité est le suivant : chiffres < lettres majuscules < 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	

Note :

On peut ignorer la lors de la comparaison grâce à la commande `compareToIgnoreCase()`

Exemple. Comparaison en ignorant la case

```
1 "ABc".compareToIgnoreCase("ABZ")
2 // Vaut 0 puisque "abc" == "abc"
```

Syntaxe. Déterminer l'index d'une sous-chaine

On peut utiliser la comande `.indexOf()` pour identifier la position à laquelle on trouve une sous-chaine X. Si la chapine ne contient pas la sous-chaine recherchée, la commande **retourne -1**.

Exemple.

```
1 /* Soit une chaine ; pour trouver la positon de X = jour
2 dans la chaÃ@ne on utilise : */
3 "Bonjour Monsieur!".indexOf("jour")
4 // Vaut 3 (int); jour dÃ@bute Ã la position 3 du String.
5 "Bonjour Monsieur!".indexOf("jours")
6 // Vaut -1 ; la sous-chaine n'existe pas
```

Note :

On peut spécifier la position à partir de laquelle il faut chercher l'index en fournissant *second paramètre* à la commande `String.indexOf()`

Exemple. Trouver l'index en spécifiant le début de la sous-chaine

```
1 "Bonjour Monsieur!".indexOf("on",0)
2 //Vaut 1
```

```

3      "Bonjour Monsieur!".indexOf("on", 5)
4      // Vaut 9 ; le premier "on" trouve a partir de la position 5
5      "Bonjour Monsieur!".indexOf("on", 10)
6      // Vaut -1 ; "on" est introuvable apres la position 10.
7

```

Note :

On peut également utiliser un `char` à la place d'un `String` pour effectuer la recherche. Exemple
`"Bonjour".indexOf('j') ↔ "Bonjour".indexOf("j")`

Syntaxe. Convertir un type primitif en entier

On **ne peut pas** utiliser de cast. Il faut utiliser la formule `valeur + ""`. Alternativement, on peut utiliser la commande `String.valueOf()`

Exemple. Convertir un type primitif en entier

```

1      10 + "" // Vaut "10"
2      128.7 + "" // Vaut "128.7" et est de type String
3
4      String.valueOf(10) // Vaut "10"
5      boolean a = true;
6      a + "" // Vaut "true"
7      String.valueOf(a) // Vaut "true"

```

Syntaxe. Convertir un String représentant un nombre en un entier

Les commandes `TypeVoulu.parseTypevoulu` permet la conversion `String` de nombre *rightarrow* `int`.

Exemple. Conversion String de nombre → int

```

1      Integer.parseInt("1" + "2") // Vaut 12 et est de type int
2      Double.parseDouble("-3e-1") // Vaut -0.3 et est de type Double.
3      Integer.parseInt("bonjour") // Engendre un erreur a l'exécution

```

2.5 Entrées et Sorties

Syntaxe. Affichage à l'écran

On utilise `System.out.println` pour effectuer un saut à la ligne lors de l'affiche. La commande `System.out.print` affiche *sans saut à la ligne*.

Syntaxe. Interaction avec l'utilisateur

La commande `args` permet d'enregistrer et manipuler des arguments *de la ligne de commande*. La commande `Scanner` permet la lecture de données interactives.

Exemple. Utilisation d'arguments en ligne de commande

```

1 public class Hello {
2     public static void main(String[] args) {
3         // Le code va ici
4         System.out.println("Hello, " + arg[0] + " !");
5     }
6 }

```

Note :

la variable args est un tableau de Strings. Chaque argument présenté au lancement du programme est un *String* qui est enregistré dans la variable args.

Exemple.

```

1 public class Max {
2
3     public static void main(String[] args) {
4         int max = -1
5
6         for (int i=0, i<args.length; i++) {
7             max = Math.max(max, Integer.parseInt(args[i]));
8         }
9
10        System.out.println("Maximum=" + max);
11    }
12 }
13 /* Execution :
14 $ javac Max.java
15 $ java Max 1 6 3 1 5 3
16 Maximum=6
17 */

```

sectionModèle Mémoire

Concept. Mémoire d'un programme

Elle est conceptuellement séparée en deux parties : *la pile* et *le tas*.

Définition 2.1: test