

# Programmation en Java : Envoi n°1 - Partie 1

---

05 janvier 2026

Passage de Python à Java

Types primitifs en Java

Boucles, conditions et fonctions

## Passage de Python à Java

# Passage de Python à Java

En Python, vous avez probablement déjà vu ou codé quelque chose comme ceci :

```
1 def double(n):  
2     return n * 2
```

```
1 >>> myVar = 10  
2 >>> is_even(myVar)  
3 True
```

Il vous suffit de donner un nombre (de type **int** pour entier ou **float** pour un nombre décimal) à la fonction pour obtenir un **boolean** (**True** ou **False**) qui vous indique si votre nombre est pair.

# Passage de Python à Java

Et si je décide d'attribuer une autre valeur à ma variable :

```
1 def double(n):  
2     return n * 2
```

```
1 >>> myVar = "Toto"  
2 >>> is_even(myVar)  
3 TotoToto
```

```
1 >>> myVar = ["Toto", "Tata"]  
2 >>> is_even(myVar)  
3 ["Toto", "Tata", "Toto", "Tata"]
```

Ici, un comportement potentiellement inattendu résulte de notre fonction.

# Passage de Python à Java

Et si je décide d'attribuer encore une autre valeur à ma variable :

```
1 def double(n):  
2     return n * 2
```

```
1 >>> myVar = {"Lionel": "64", "Gabriel": 22}  
2 >>> is_even(myVar)  
3 ...  
4 return n * 2  
5     ~~    ~~  
6 TypeError: unsupported operand type(s) for *: 'dict' and 'int'
```

Ici, une erreur s'est produite à l'exécution car l'opération n'est pas possible avec une variable de type **str** : un dictionnaire ne peut pas être multiplié par un nombre.

## Passage de Python à Java

C'est pour éviter ce genre d'erreur qu'en Java, on déclare le type de nos variables sans quoi le code ne compilera pas. Le code python :

```
1 def is_even(n):  
2     return n % 2 == 0
```

... va devenir en Java :

```
1 static int letsDouble(int n){ // "double" est un nom reserve  
2     return n * 2;  
3 }
```

Ici, on peut aisément voir le type de variable accepté en entrée (int), ainsi que le type de variable en sortie (int). Pour le moment, nous n'entrerons pas dans les détails de la signification de "static" que nous verrons plus tard lorsque nous aborderons la notion d'objet.

# Passage de Python à Java

En **Python**, on dit que le **typage est dynamique**. Vous pouvez attribuer une valeur numérique à une variable, puis lui attribuer ensuite une chaîne de caractères comme valeur. Le typage change dynamiquement suite à la nouvelle valeur assignée.

```
1 >>> x = 8
2 >>> print(x)
3 8
4 >>> type(x)
5 <class 'int'>
6 >>> x = "toto"
7 >>> type(x)
8 <class 'str'>
```

## Passage de Python à Java

Au contraire en **Java**, on dit que le **typage est statique**. Le type de la variable est défini au moment de sa déclaration. Ce type définit les valeurs que peut prendre la variable. **Le type de la variable ne peut plus être modifié par la suite.**

```
1 int x = 8; // Peut se faire en 2 temps : int x; x=8;  
2 System.out.println(x); // Equivalent du print(), affiche 8  
3 x = "timoleon"; /* Ne compile pas/interdit ! */
```

Vous noterez dans l'exemple ci-dessus qu'en Java **les commentaires** (indications ignorées à la compilation et l'exécution) se déclenchent par le symbole `//` ou sont encadrés de `/* ... */` pour des commentaires multilignes (au lieu du `#` et des `""" ... """` de Python)

# Passage de Python à Java

## Python

- les variables, arguments, et sorties n'ont pas de type prédéfini
- le type de la VALEUR attribuée à une variable peut changer
- ce typage dynamique peut être à l'origine d'erreurs lors de l'exécution du code.
- dans des programmes longs, un typage dynamique peut complexifier la compréhension du code par le lecteur

## Java

- les variables, arguments, et sorties ont un type prédéfini : déclaré et non-modifiable.
- lors de la compilation du code (avant l'exécution), des vérifications de compatibilité des types sont effectuées.
- dans des programmes longs, un typage statique permet au lecteur de comprendre plus facilement le code, comme le rôle des variables et des méthodes (ou fonctions).

## En Python comme en Java - conventions et bonnes pratiques

- le code doit être commenté : vous serez plus souvent amenés à reprendre le code de quelqu'un d'autre qu'à retravailler votre propre code (*Ne faites pas à autrui, ...*), et même dans ce cas... *"Où l'intérêt cesse, se perd aussi la mémoire"*, Goethe
- les variables, arguments, classes/objets doivent avoir des noms explicites, sans être trop longs (abréviations), et en anglais (pour éviter les caractères accentués)
- ces noms doivent forcément commencer par une lettre, qui peut être complétée par des lettres, chiffres et certains caractères non-réservés (*isJavaIdentifierPart(char)* pour tests) pour séparer les noms "composés" comme la notation *CamelCase*.

```
1 int dateOfBirth; // Chameau
2 int date_birth = date$birth; // autres
```

- la convention en Java est de n'utiliser une **majuscule en début de nom QUE pour les classes d'objets**.

# Types primitifs en Java

# Types primitifs en Java

En programmation objet, on différencie les **types primitifs** des **types d'objets** :

- Les **types primitifs** (boolean, int, double, char...) contiennent directement une valeur simple. Ils attribuent une valeur par défaut aux variables instanciée (false, 0, 0.0, 'A', '\u0000', ...). Ils n'ont pas de méthodes/fonctions associées.
- Les **types objets** : sont des modèles/moules sur lesquels on crée des objets appelés **instances de la classe**. L'objet bénéficie alors de méthodes/fonctions prédéfinies dans la classe. Par exemple, les chaînes de caractères (String) peuvent être converties en majuscules.

Pour commencer, nous allons nous concentrer sur les types primitifs, avant d'utiliser des classes/objets (et leurs méthodes) prédéfini(e)s en Java, puis de développer nos propres objets/classes, plus tard dans ce cours.

# Types primitifs en Java

En Java, il existe des types dits **primitifs** :

Catégorie	Type	Intervalle de valeurs
Booléen	boolean	0 ou 1 = false ou true
Entiers	byte	-128 à 127
	short	-32 768 à 32 767
	int	-2 147 483 648 à 2 147 483 647
	long	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
Réels	float	$\pm 3.4 \times 10^{38}$ , précision $1.4 \times 10^{-45}$
	double	$\pm 1.7 \times 10^{308}$ , précision $1.4 \times 10^{-45}$
Texte	char	'\u0000' à '\uFFFF' (dont 'a', 'b', ..., 'Y', 'Z')

# Types primitifs en Java

Attention à ne pas confondre char String et str !

## char (Java)

- représente **un seul** caractère unicode
- type **primitif**
- **pas de concaténation** directe (sauf conversion en String).

## String (Java)

- représente **une** séquence de caractères
- type **objet** (java.lang.String)
- peut être concaténé :  
`"A" + "B" → "AB"`

## str (Python)

- représente **une** séquence de caractères
- Type **objet** natif (str)
- peut être concaténé :  
`"A" + "B" → "AB"`

# Types primitifs en Java

Attention à quelques cas :

```
1 float a = 3.14f; // Sans suffixe, erreur de compilation !
2 int b = 2_147_483_647; // Possible pour rendre plus lisible
3 System.out.println(b+1); // Affiche -2147483648
4 int c = 'A'; // -> c = 65
5 char d = 65; // -> d = 'A'
```

Les *char* sont des cas particuliers,  
pouvant également être considérés  
comme des entiers entre 0 et 65535 :  
leur position dans la table ASCII.

	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89	90	91 lsqb	92 bsol	93 rsqb	94 hat	95 lowbar
	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^
												LEFT SQ. BRACKET	REVERSE SOLIDUS	RT. SQ. BRACKET	CIRCUM'X ACCENT
96 grave	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n
GRAVE ACCENT															
112	113	114	115	116	117	118	119	120	121	122					
	p	q	r	s	t	u	v	w	x	y	z				

# Types primitifs en Java

Les valeurs sont converties implicitement, tant que l'on ne perd pas d'information

```
1 double e = 3; // la valeur stockee est "completee" -> c = 3.0
2 int f = 3.0; // Sanction : erreur de compilation
3 double g = 0.1 * 3.0; // Rare bizarrerie : 0.30000000000000004
4 float h = 0.1f * 3.0f; // OK : h = 0.3
```

On peut sinon forcer la conversion d'une valeur en plaçant le type souhaité entre parenthèse avant celle-ci, au risque de perdre de l'information. Par exemple :

```
1 int i = (int) 2.6; // Valeur flottante convertie en int -> i=2
2 double j = 1/2; // division d'entiers -> j = 0.0
3 double k = 1/((double)2) // division de reals -> k = 0.5
```

Le type d'une opération (ou expression arithmétique) dépend des types qui entrent en jeu dans cette opération : elle retourne le plus générique !

# Types primitifs en Java

Le type d'une opération (ou expression arithmétique) dépend des types qui entrent en jeu dans cette opération : elle retourne le plus générique !

Le résultat est **de type int** lorsque **toutes les valeurs sont de type int**. S'il y a un double, le résultat sera **de type double**

Opérateur	Exemple	Opérateur	Exemple
int + int => int	3 + 2 = 5	int + double => double	3 + 8.6 => 11.6
int - int => int	2 - 3 = -1	double - int => double	8.6 - 5 => 3.6
int * int => int	2 * 3 = 6	int * double => double	2 * 3.6 => 7.2
int / int => int	14 / 3 = 4	double / int => double	2.5 / 2 => 1.25
int % int => int	14 % 3 = 2	double % int => double	3.01 % 2 => 1.0099999999999999

Encore une fois, les réels donnent parfois des résultats surprenants...

# Types primitifs en Java

Il est possible de combiner opérateurs (+, -, \*, / et %) avec l'affectation, par exemple :

```
1  l=2; l+=10; // Equivalent a : a=a+10, renvoie 12
```

Des opérateurs similaires existent pour modifier les nombres en binaire(<<, >>, >>>)

```
1  m=5; m<<=3 ; /* 5=>101 en binaire, apres decalagee de trois 0 a  
2  droite devient 10100=36+4=40. >> conserve le signe, pas >>> */
```

Il existe également des opérateurs d'incrémentation ou décrémentation (++/-) qui s'appliquent avant ou après l'affectation (et dans d'autres contextes), tels que :

```
1  n=5; o=++n; // n s'incrémente puis sa valeur affectée -> o=n=6  
2  p=3; q=p--; // p affectée à q, puis p décrémente -> q=3 et p=2  
3  r=r++; // attention : r ne change pas de valeur !
```

# Types primitifs en Java

Les opérateurs logiques (>, >=, <, <=, == et !=) permettent de comparer des nombres et/ou caractères et renvoie une valeur booléenne :

```
1 a=3; b=3.0; System.out.println(a==b) // Affiche "true"
```

Ces comparaisons logiques, et donc leurs résultats, peuvent être combinés par les opérateurs binaires *and* (&&), *or* (||) et l'opérateur unaire *not* (!).

Important : comme en mathématiques et en logique, il existe un ordre de priorité entre tous les opérateurs, qui ne peut être modifier que grâce à l'ajout de parenthèses (cf <https://blog.paumard.org/cours/java/chap05-noms-operateurs-operateurs.html>)

```
1 int a=0;
2 // Quel est le resultat de l'expression ?
3 2 * 7 + 12 >= 28 - 1 / 2.0 && ! true || a++ == 1
```

## Boucles, conditions et fonctions

# Boucles

Ces structures de contrôle permettent la répétition d'instructions.

Ci-dessous, deux cas où si la condition n'est pas remplie, alors les instructions du bloc ne sont pas réalisées :

```
1  for( initialisation ; condition ; modification ) {  
2  // Bloc d'instructions a repeter  
3  }  
4  
5  // Boucle precedee de l'initialisation du "for"  
6  while(condition) {  
7  // Bloc d'instructions a repeter, avec la modification du "for"  
8  }
```

Exemple : initialisation serait `int i=0`, condition serait `i<5` et modification serait `i++`

# Boucles

Ces structures de contrôle permet la répétition d'instructions : Cas où la condition n'est testée qu'après une première réalisation des instructions

```
1 do {  
2 // Bloc d'instructions a repeter  
3 } while(condition);
```

Quelle que soit le genre de boucle, il est possible d'inclure deux autres types de contrôles depuis l'intérieur :

- **break** : interrompt immédiatement la boucle
- **continue** : passe immédiatement à l'itération suivante de la boucle

# Conditions

La structure de contrôle *if* permet de spécifier des instructions à effectuer si un test/une condition est remplie

```
1 if (condition) {  
2     // bloc "test réussi"  
3 }
```

Optionnellement, on peut compléter par la structure *else* pour indiquer des instructions alternatives, en cas d'échec du test :

```
1 if (condition) {  
2     // bloc1 "test succes"  
3 } else {  
4     // bloc2 "test echec"  
5 }
```

# Conditions

L'opérateur ternaire est une **alternative compacte** :

```
1 // condition ? blocVrai : blocFaux
2 while(a!=1)
3     a = a % 2 ? a / 2 : a * 3 + 1
```

Quel est le résultat/affichage du programme suivant ?

```
1 int number = 20; int tentatives = 0;
2 while(nb != 1) {
3     tentatives += 1 ;
4     if(nb%2==1) { nb*=3; nb+=1; }
5     else { nb/= 2; continue; }
6     if(nb == 1) { break; }
7     print(number);}
8 print(tentatives)
```

# Conditions

Il existe enfin l'**opérateur *Switch***, une alternative plus lisible aux multiples *if-else* imbriqués, pour déterminer suivant le cas (*case* en anglais) quelles instructions effectuer suite au résultat d'un même test/d'une même condition :

```
1  switch (expression) {
2      // Expression peut etre un int, String, ...
3      case valeur1:
4          // Instructions si expression == valeur1
5          break;
6      case valeur2:
7          // Instructions si expression == valeur2
8          break;
9      ...
10     default:
11         // Instructions si aucune valeur ne correspond
12 }
```

# Fonctions

Chaque fonction est une recette :

- elle utilise des ingrédients, les **paramètres**
- elle les manipule/transforme en suivant les **instructions**, ponctuées de structures de contrôle (**conditions et boucles**)
- elle produit un unique **résultat**

Rappel de la syntaxe :

```
1 public class MyClass {  
2     static char myFunction(int param1, float param2){ /* DEF */ }  
3     // "void" si absence de "return"  
4     public static void main(String[] args) {  
5         int a=1; float b=3.14f;  
6         char c = myFunction(a, b) /* CALL */ }  
7     }
```

## Retour sur la comparaison Python vs. Java

Ces instructions comportent des différences notables de syntaxe entre Java et Python

- En Python il était possible d'écrire plusieurs instructions sur une seule ligne à condition de les séparer par un point-virgule. En Java également, mais **toute instruction doit être conclue par un ";"**. Il est cependant désormais possible de les étaler sur plusieurs lignes également : le système attend la conclusion.

```
1  int
2  myX
3  =
4  21
5  ;
6  System.out.println(
7  myX
8  ); // Ca fonctionne !
```

## Retour sur la comparaison Python vs. Java

- *True* et *False* ne prennent plus de majuscule et deviennent ***true*** et ***false***
- les *if-else*, *boucles* et définition de fonction ne requiert plus les ":"
- L'indentation n'est plus obligatoire (mais fortement recommandée). Il devient cependant nécessaire d'utiliser des accolades pour définir des blocs d'instructions. Sinon, une seule des instructions suivantes sera effectuée.

```
1  if(true) {System.out.println(1);  
2      System.out.println(2);  
3      System.out.println(3);  
4  System.out.println(4);}  
5  // Affichera les 1, 2, 3 et 4  
6  if(false)  
7      System.out.println(1);  
8      System.out.println(2);  
9  // N'affichera que 1
```